TRAINS

19971007 106

The Design and Implementation of the TRAINS-96 System: A Prototype Mixed-Initiative Planning Assistant

Goerge M. Ferguson, James F. Allen,
Brad W. Miller, and Eric K. Ringger

TRAINS Technical Note 96-5
October 1996

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

The Design and Implementation of the TRAINS-96 System: A Prototype Mixed-Initiative Planning Assistant

George Ferguson
James F. Allen
Brad W. Miller
Eric K. Ringger

The University of Rochester
Computer Science Department
Rochester, New York 14627

TRAINS Technical Note 96-5

October 1996

Abstract

This document describes the design and implementation of TRAINS-96, a prototype mixed-initiative planning assistant system. The TRAINS-96 system helps a human manager solve routing problems in a simple transportation domain. It interacts with the human using spoken, typed, and graphical input and generates spoken output and graphical map displays. The key to TRAINS-96 is that it treats the interaction with the user as a *dialogue* in which each participant can do what they do best. The TRAINS-96 system is intended as both a demonstration of the feasibility of realistic mixed-initiative planning and as a platform for future research. This document describes both the design of the system and such features of its use as might be useful for further experimentation. Further references and a comprehensive set of manual pages are also provided.

This material is based upon work supported by ARPA – Rome Laboratory under research contract no. F30602-95-1-0025, by the Office of Naval Research under research grant no. N00014-95-1-1088, and by the National Science Foundation under grant no. IRI-9623665. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REPORT DOCUMENTATION PAGE

Form Approved

OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
October 1996

3. REPORT TYPE AND DATES COVERED
technical report

4. TITLE AND SUBTITLE
The Design and Implementation of the TRAINS-96 System

5. FUNDING NUMBERS
ARPA/Rome Lab F30602-95-1-0025
ONR N00014-95-1-1088

6. AUTHOR(S)
G.M. Ferguson, J.F. Allen, B.W. Miller, and E.K. Ringger

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES
Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)
Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
TRAINS TN 96-5

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
(see title page)

14. SUBJECT TERMS
mixed-initiative planning; dialogue systems; interactive systems

15. NUMBER OF PAGES
164 pages

16. PRICE CODE
free to sponsors; else $7.00

17. SECURITY CLASSIFICATION OF REPORT
unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
unclassified

20. LIMITATION OF ABSTRACT
UL

Contents

1 Introduction **1**

2 Overview **3**

2.1 Comparison to TRAINS-95 3

2.2 TRAINS-96 Functional Overview 3

3 Intermodule Communication: KQML **6**

3.1 KQML Basics 6

3.2 KQML Syntax 6

3.3 KQML Parameters 7

3.4 KQML Performatives 8

3.5 KQML in TRAINS-96 9

3.6 KQML Summary 10

4 Input Manager **11**

4.1 Client Registration 11

4.2 Selective Broadcast 11

4.3 Non-selective Broadcast 13

4.4 Module Status 13

4.5 Module Classes 14

4.6 Input Manager Display 14

5 Process Manager **16**

5.1 Starting Processes 16

5.2 Input Manager Connection 16

5.3 Remote Processes 17

5.4 Other Process Manager Functions 17

6 Audio Manager **18**

7 Speech Recognition **19**

7.1 Speech Recognizer Operation 19

7.2 Speech Recognizer Files 20

8 Speech Post-Processor **21**

8.1 Description 21

8.2 Speech Post-Processor Models 21

8.3 Speech Post-Processor Operation 21

9 Speech Generation **24**

10 Keyboard Manager **26**

11 Display **27**

11.1 Object Manipulation Messages 27
11.2 Object Highlighting Messages 31
11.3 Dialog Box Messages 31
11.4 Display Control Messages 31
11.5 Display Output Messages 31

12 Parser **33**

12.1 Parser Implementation 33
12.2 Parser Input 33
12.3 Parser Operation 33
12.4 Parser Output 36

13 Discourse Manager **39**

13.1 Discourse Manager Implementation 39
13.2 Discourse Manager Operation 40

14 Problem Solver **42**

14.1 Problem Solver Principles 42
14.2 Problem Solver Representation 43
14.3 Problem Solver Implementation and Operation 45

15 Other Modules **51**

15.1 Speech Controller (SPEECH-X) 51
15.2 Startup Screen (SPLASH) 51
15.3 Transcript (TRANSCRIPT) 53
15.4 Scenario Chooser (SCENARIO) 53
15.5 Shortcuts Panel (SHORTCUT) 54
15.6 Sound Effects (SFX) 55
15.7 Parse Tree Viewer (PVIEW) 55
15.8 Replay 56
15.9 Input Manager Utilities 58
15.10Dialog Archiving Tools 59

A Running the TRAINS System **62**

A.1 Setup Environment 62

A.2 Start IM and PM 62

A.3 Start Other Modules 63

B Speech Lab Setup **64**

B.1 Introduction 64

B.2 Audio Rack Setup 64

B.3 Audio Rack Settings 66

C Travel System Instructions **67**

C.1 System Overview 67

C.2 Before You Go 67

C.3 Workstation Setup 68

C.4 Workstation Boot Procedure 68

C.5 Audio Setup 70

C.6 Troubleshooting 73

C.7 Shutdown 73

C.8 Shipping List 74

D Manual Pages **76**

D.1 `dlg_check`: Check TRAINS-96 dialogue contents 77

D.2 `dlg_org`: Organize TRAINS-96 dialogue contents 78

D.3 `dlg_play`: Play TRAINS-96 dialogue utterances 80

D.4 `taudio`: TRAINS Audio Manager 81

D.5 `tdisplay`: TRAINS Display Module 84

D.6 `tim`: TRAINS Input Manager 91

D.7 `tim_cat`: Send KQML messages from stdin to TRAINS IM 96

D.8 `tim_client`: Send and receive KQML messages to/from TRAINS IM 97

D.9 `tim_exec`: Exec a program with stdin/stdout connected to TRAINS IM . . 98

D.10 `tim_msg`: Send a KQML message to TRAINS IM 99

D.11 `tkeyboard`: TRAINS Keyboard Manager 101

D.12 `tparser`: TRAINS Parser module 105

D.13 `tpm`: TRAINS Process Manager 108

D.14 `tpsm`: TRAINS Problem Solving module 111

D.15 `tpview`: TRAINS Parse Tree Viewer 113

D.16 `trains`: Run the TRAINS System 115
D.17 `treplay`: Replay a TRAINS System session 117
D.18 `tscenario`: TRAINS Scenario Chooser 119
D.19 `tsfx`: TRAINS Sound Effects module 121
D.20 `tshortcut`: TRAINS Shortcut Panel 123
D.21 `tspeech`: TRAINS version of Sphinx-II speech recognizer 125
D.22 `tspeechpp`: TRAINS Speech Post-Processor 129
D.23 `tspeechx`: TRAINS Speech Controller 133
D.24 `tsplash`: TRAINS Splash Screen module 137
D.25 `ttc`: TRAINS TrueTalk client using AudioFile server 141
D.26 `ttcl`: TRAINS Discourse Manager module 144
D.27 `ttranscript`: TRAINS Transcript module 148
D.28 `tts`: Runs TrueTalk server 151
D.29 `tttalk`: TRAINS Speech Generation module 152
D.30 `libKQML`: TRAINS System KQML Library 154
D.31 `libtrlib`: TRAINS System Module Library 158
D.32 `libutil`: TRAINS System Utility Library 161

List of Tables

1 Reserved KQML Parameters 7
2 Common KQML Performatives 8
3 TRAINS Standard Messages 9
4 TRAINS Error Performatives 9
5 Input Manager (IM) Messages 12
6 Input Manager Status Indicators 14
7 Process Manager (PM) Messages 17
8 Speech Recognizer (SPEECH-IN) Messages 20
9 Speech Post-Processor (SPEECH-PP) Messages 22
10 Speech Post-Processor Models 22
11 Speech Generator (SPEECH-OUT) Messages 25
12 Display (DISPLAY) Messages 29
13 Display Object Creation Attributes 30
14 Display Object-specific Creation Attributes 30
15 Display Output Messages 32
16 Parser Speech-Act Types 35
17 Possible constraints for action `GO` 44
18 Possible constraints for action `STAY` 44
19 Problem Solver (PS) Interpretation Requests 46
20 Problem Solver (PS) Update Requests 48
21 Problem Solver (PS) Knowledge Base Requests 49
22 Speech Controller (SPEECH-X) Messages 52
23 Transcript (TRANSCRIPT) Messages 53

List of Figures

1 TRAINS-96 Schematic Diagram 4
2 KQML BNF syntax 7
3 Input Manager Classes in TRAINS-96 15
4 Input Manager Display 15
5 SPEECH-OUT Functional Diagram 25
6 Sample DISPLAY module map display 28
7 Sample Parser Output 37
8 TRAINS-96 Speech Controller Display 52
9 Sample Parse Tree Viewer Display 57
10 Speech lab audio rack front and back panels 65
11 Travel system connection diagram 69
12 Travel system audio connection diagram 71
13 Travel system packing diagram 75

1 Introduction

The TRAINS Project at the University of Rochester is a long-term research effort to develop intelligent planning assistants that interact with their human managers using natural language. The guiding principle of our approach is that human-computer interaction should be treated as a *dialogue* between the participants, each of whom brings different skills and objectives to the conversation. The goal is to allow each participant to do what they do best. For the human, this usually means providing fairly high-level strategic advice, making important decisions, and delegating authority as needed. The computer assistant's strength is usually in managing the low-level details, alerting the human to problems or possibilities, and developing exploratory scenarios for human and joint evaluation. The result of this approach is truly *mixed-initiative* interaction—both participants can guide the conversation as needed to accomplish their goals.

Some of the many research issues being pursued in this work are:

- Representation of complex, realistic domains
- Integration of spoken, written, and graphical (*i.e.*, mouse- or pen-based) input within a language-oriented framework
- Maintaining the context of the interaction, including shared background, focus of attention, *etc.*
- Robust understanding, especially of spoken language in the presence of errors
- Use of speech generation and graphical displays, especially maps, for human-computer interaction
- Explicit representation of and reasoning about the problem solving process
- Integration of multiple, independent specialized reasoners
- Trade-offs between expensive "guaranteed" algorithms and faster "satisficing" methods in the context of a mixed-initiative system
- Distributed processing with expressive inter-module communication
- Evaluation of interactive systems using end-to-end task-based metrics

Considerably more detail regarding these and other aspects of the research can be found at our WWW site:

`http://www.cs.rochester.edu/research/trains/`

Most papers related to the TRAINS Project are either available directly from this site or are listed in a comprehensive bibliography.

For the past several years, we have periodically implemented our approaches to these problems as they then stand. The result has been a succession of interactive planning

assistants, dubbed the TRAINS Systems [Allen and Schubert, 1991; Allen *et al.*, 1995; Heeman and Allen, 1995; Ferguson *et al.*, 1996]. The most recent incarnation of the system, TRAINS-96, is described in this report. We believe strongly that this process of bootstrapping our research using an actual running system, however limited, results in a deeper understanding of the problems than would theorizing based on introspection. As well, some of the issues, such as distributed systems and evaluation, really cannot be investigated without a running system from which to start. And finally, the fact is that it's more fun to sit down and talk to the system than to run thousands of experiments on simulated data and sift through the answers. Both the successes and the shortcomings are readily apparent and lead to refinement of the theories and improvements in the next generation of the system.

The plan for the remainder of this document is as follows. First, we give an overview of the TRAINS-96 system, including some of the history of the project as it pertains to the design goals for TRAINS-96. Next, we describe the intermodule communication language used in TRAINS-96, which is based on KQML, the Knowledge Query and Manipulation Language. Subsequent sections describe each of the modules of the TRAINS-96 system. Appendices provide information about setting up and running the TRAINS-96 system, as well as a comprehensive set of manual pages.

2 Overview

As in TRAINS-95, the TRAINS-96 System is an interactive planning assistant in a simplified transportation domain. The user is presented with a map of, say, the northeast United States, and is given a task of routing a set of engines to various cities on the map. During the interaction, a variety of environmental problems can occur that neither the user nor the system was aware of in advance. The user can communicate using either spoken or typed text, as well as clicking on display objects. The system communicates using spoken language, map displays, and dialog boxes.

2.1 Comparison to TRAINS-95

Compared to TRAINS-95, the TRAINS-96 system is better designed, more robust, and, we believe, better at helping the user with the routing task. In terms of design, TRAINS-96 uses a communication infrastructure based on KQML, the Knowledge Query and Manipulation Language (Section 3). This formalizes many aspects of the system that were previously *ad hoc*, and simplifies the implementation of many of the modules and tools. As well, all modules have been redesigned, generally resulting in smaller modules, each with a simpler role. For example, the monolithic Lisp program at the heart of TRAINS-95 has been broken into several separate modules, each of which can be implemented, tested, and used independently. In terms of robustness, the system has run literally hundreds of sessions with dozens of users. Both the internal robustness of the system (*i.e.*, avoiding crashes) and its external robustness (*e.g.*, to user speech errors or system limitations) have been continually improved. We have left the system running for hours unattended while in use. Unfortunately, the task itself remains very simple, as it was in TRAINS-95. We will be conducting a set of experiments on the efficacy of the system, as we did for TRAINS-95 [Sikorski and Allen, 1996], and we believe that the improvements will be evident. Nonetheless, making the task more complex is our top priority for the next phase of the TRAINS Project.

2.2 TRAINS-96 Functional Overview

The TRAINS-96 System is composed of a set of independent modules connected by a message-passing communication infrastructure arranged in a star topology. In most cases, each module is implemented as a separate Unix process, although there is nothing to prevent a single process from embodying several modules, or a single module from using several processes. Indeed, both possibilities are used in TRAINS-96 (the former by the Discourse Manager, Section 13; the latter by modules that access external servers, such as Speech Recognition and Generation, Sections 7 and 9). The processes making up the system are connected by an Input Manager process (Section 4) and managed by a Process Manager process (Section 5).

To illustrate the organization of a typical configuration of the TRAINS-96 system, consider the schematic diagram shown in Figure 1. Note that neither the Input Manager nor the Process Manager nor a variety of other supporting modules are shown in the figure. Arrows in the figure illustrate the flow of information through the system (actual communication takes place via the Input Manager).

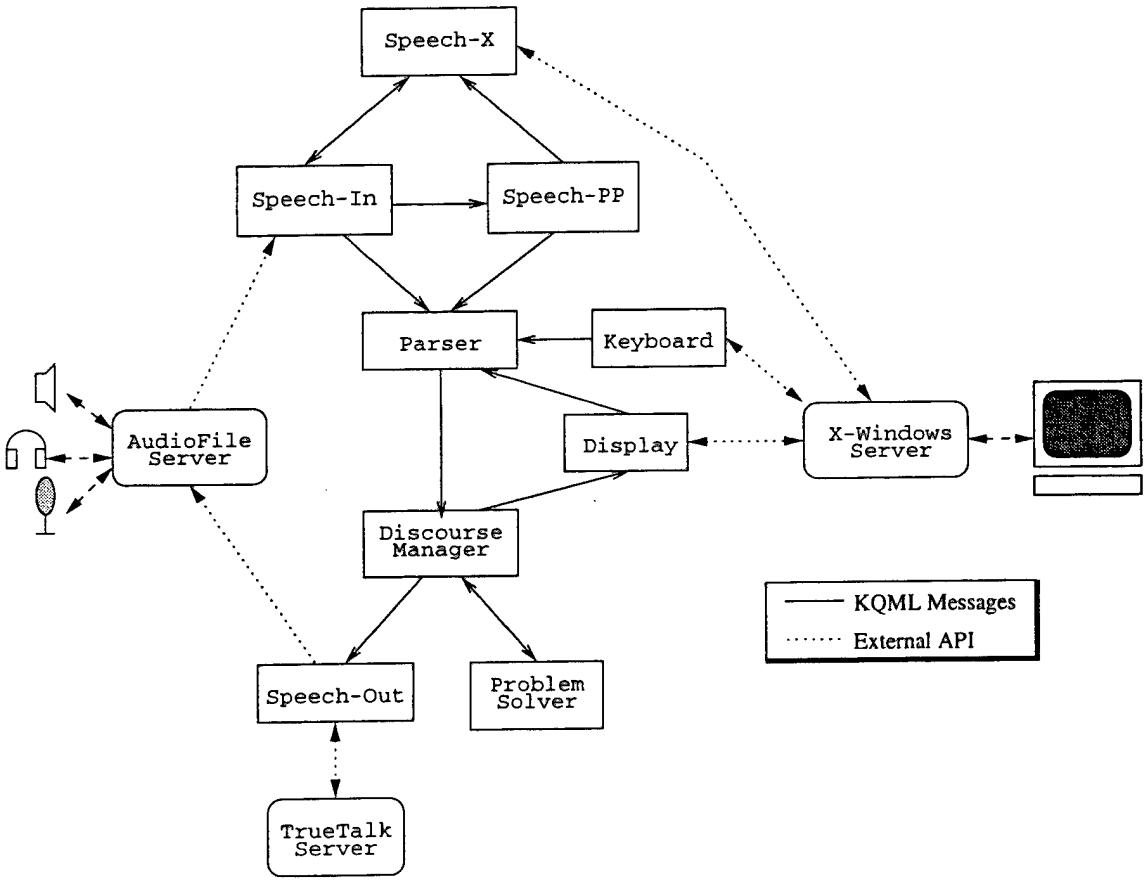

Figure 1: TRAINS-96 Schematic Diagram (Input Manager, Process Manager, *etc.*, not shown)

Starting at the top of the figure, the user might start their utterance by pressing the "Click to Talk" button on the Speech controller, Speech-X (Section 15.1). This sends a message to the Speech-In module (Section 7) requesting that it start recognition. Speech-In then reads audio data from the AudioFile server[1] and outputs a series of messages corresponding to recognized words. These messages are received by both the Parser (Section 12) and the Speech Post-Processor (Section 8), the latter of which also outputs words to the Parser. Both speech modules' outputs are monitored and displayed by the Speech-X module.

When the user releases the "Talk" button (or otherwise indicates they are done talking), the Parser outputs a logical form summarizing the content of the user's utterance, as well as such factors as its confidence in the interpretation. This is picked up by the Discourse Manager (Section 13), which uses a rule-based system to determine what the user meant and what to do about it. It uses the Problem Solver (Section 14) to help interpret the utterance in terms of the plan or plans currently under consideration. The Problem Solver manages the resources of the specialized domain reasoners at the system's disposal—in TRAINS-96 this is simply a specialized route planner built into the Problem Solver.

Finally, once the Dialogue Manager has settled on an interpretation and decided what to do in response, speech output is produced by sending messages to the Speech-Out module (Section 9), which uses an external server to generate the audio and then passes it to the AudioFile server for playback. Map displays are generated by sending messages to the Display module (Section 11), which renders objects onto the X-Windows display.

As mentioned previously, this brief description omits many of the complexities of the TRAINS-96 System. It does not describe the infrastructure that supports the modules execution and communication. As well, there are facilities for logging, debugging, and replay, all of which are essential to actual everyday use of the system. Many of these details are described in subsequent sections of this report.

[1]The AudioFile server and API provide network-transparent access to and sharing of audio resources, similar to what the X Window System does for graphical resources. See Section 6.

3 Intermodule Communication: KQML

Inter-module communication in the TRAINS System is accomplished by modules exchanging messages expressed using KQML, the Knowledge Query and Manipulation Language. The reasons we have chosen KQML for use in TRAINS are:

- Previously, TRAINS modules exchanged ASCII strings whose format was defined on an *ad hoc*, module by module basis. KQML imposes a regular structure on the messages (which are still sequences of ASCII characters), making it easier to write sharable, reusable routines for manipulating them.
- KQML defines structure without restricting content. Aside from some very general syntactic constraints, KQML is designed to be independent of the details of the messages being exchanged. Certain conventions are used to permit efficient communication, but the meanings of the messages are defined independently.
- KQML was designed to support knowledge-based systems such as TRAINS. While we do not necessarily agree with all the KQML prescriptions, we believe that our use of KQML makes future integration with other components more feasible.

The rest of this section describes KQML and its use in the TRAINS System. It is intended as a complement to the KQML specification [Finin *et al.*, 1993], which provides more details regarding message formats. Subsequent sections will describe the messages understood by the components of the TRAINS System.

3.1 KQML Basics

KQML defines messages in terms of "performatives," a term from the semantics of natural language verbs. In natural language, a performative is an utterance that accomplishes something by the very fact of its being uttered, for example greeting someone by saying "hello." In KQML, the various message types are identified with such verbs, and the message is referred to as a performative. We will use the terms "message" and "performative" interchangeably, and also use the term "performative" to refer to the verb involved (as opposed to its arguments). Examples of KQML performatives include "`tell`," "`ask-one`," "`reply`," *etc.* These will be described in more detail below.

3.2 KQML Syntax

The syntax of a KQML message is Lisp-like, based on parenthesized lists of tokens and strings. A simple BNF grammar of KQML syntax is shown inFigure 2. Tokens are roughly sequences of non-space characters; strings are roughly double-quoted sequences of characters. The class of tokens beginning with a colon (":") are called "keywords" and typically have special meanings. A KQML performative is a list whose first token is the verb (name) of the performative, followed by keyword-value pairs defining the "parameters" of the performative. For example, the following is a `tell` performative with two arguments, the receiver of the message and its content:

```
<performative> ::= ( <word> {<white> :<word> <white> <expr>}* )
<expr> ::= <word> | <quotation> | <string> |
           (<word> {<white> <expr>}*)
<word> ::= <char><char>*
<char> ::= <alphabetic> | <numeric> | <special>
<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?
<quotation> ::= '<expr> | `<comma-expr>
<comma-expr> ::= <word> | <quotation> | <string> | ,<comma-expr> |
                 (<word> {<white> <comma-expr>}*)
<string> ::= "<stringchar>" | #<digit><digit>*"<ascii>*
<stringchar> ::= \<ascii> | <ascii-not-\-or-">
```

Figure 2: KQML BNF syntax

Parameter	Type	Meaning
`:sender`	token	Identifies sender of message
`:receiver`	token	Identifies recipient of message
`:reply-with`	expr	Tag for subsequent messages
`:in-reply-to`	expr	Tag from previous message
`:content`	expr	Content of message
`:language`	expr	Syntax of content
`:ontology`	expr	Semantics of content

Table 1: Reserved KQML Parameters

```
(tell :receiver KB :content (P ?x "123"))
```

Case is usually ignored outside of strings. The KQML specification, however, is quite picky about whitespace.

3.3 KQML Parameters

To facilitate communication, the KQML specification defines seven parameters common to all performatives. While a performative needn't specify them, if they are used they must be used with the defined meanings. This makes it possible to formulate a response to a message even if most of it, even the verb, isn't understood. The seven reserved parameters are shown in Table 1.

The first four reserved parameters (`:sender`, `:receiver`, `:reply-with`, `:in-reply-to`) make communication possible by identifying the participants and allowing them to connect

Performative	Meaning
`tell`	Sender believes content to be true
`ask-if`	Sender is asking if receiver believes content to be true
`ask-one`	Sender is asking for a true instance of the content
`reply`	Sender is answering a previous message
`error`	Sender did not understand a previous message ("your fault")
`sorry`	Sender understood but could not answer a previous message ("my fault")
`request`	Sender is asking receiver to perform an action

Table 2: Common KQML Performatives

messages in logical order. The other three reserved parameters specify the content of the message, with `:language` specifying the syntax and `:ontology` the semantics. These last two parameters are not used in TRAINS-96.

3.4 KQML Performatives

KQML then goes on to define an initial set of performatives, and the specification gives an English gloss of their intended interpretation. More precise formal specification of the semantics of KQML messages is the subject of debate and research. Some of the basic performatives used in the TRAINS System are shown in Table 2, with a short description of their intended meaning. Interestingly, the `request` performative is not part of the KQML specification. Perhaps this is an artifact of KQML's focus on querying knowledge bases, rather than controlling components of a system such as TRAINS. While it is true that a request to perform action A can be expressed using the standard KQML performative `achieve` that A be performed (or perhaps that the results of A be true), this seems awkward at best. Compare:

```
(request :content (kill P))
(achieve :content (dead P))
(achieve :content (killed P))
```

In any case, we use `request` in TRAINS for the majority of the imperative commands that control modules' behaviors.

Many other KQML performatives are defined, some of which deal with naming, routing, and identification of distributed components and services. Our approach to these issues will be described in the next section, where the TRAINS-96 Input Manager is described. Suffice it to say that we have not adopted all of KQML's prescriptions in this area, which in our opinion starts to get away from KQML's focus on exchange of knowledge between knowledge-based systems.

Performative	Content	Arguments
`request`	`exit`	*status*
`request`	`chdir`	*dirstring*
`request`	`hide-window`	
`request`	`show-window`	
`tell`	`start-conversation`	`:name :lang :sex`
`tell`	`end-conversation`	

Table 3: TRAINS Standard Messages

Performative	Parameters	Meaning
`error`	`:code :comment`	`Your fault`
`sorry`	`:code :comment`	`My fault`

Table 4: TRAINS Error Performatives

3.5 KQML in TRAINS-96

In the rest of this report, the set of KQML messages understood by each module of the TRAINS-96 system will be presented in an abbreviated, tabular form. As an example, Table 3 presents the common messages understood by all modules of the system. The `exit` message requests that the module terminate with exit status as specified by the optional argument (default 0). The `chdir` message requests that the module begin using the given directory for output and logging. The `hide-window` and `show-window` requests are self-explanatory, and obviously only apply to modules with X Window displays. The `start-conversation` and `end-conversation` messages are also self-explanatory, and are typically used to initialize a module when the user starts a conversation, and to clean up when they finish. Some modules may ignore some of these messages, but they should not generate errors when received.

In addition, modules may generate a variety of messages, which will be described in the following sessions. However, two KQML performatives are used to indicate errors and should be used consistently by all modules. These are shown in Table 4. The `error` performative is used to indicate that the module did not understand a previous message. This includes ill-formed and unknown performatives, as well as those whose arguments are inappropriate. The `sorry` performative is used more sparingly to indicate that a module was unable to act on a previous message even though it understood the message and should have been able to act on it. Both performatives can provide as parameters a numeric `:code` and a string `:comment` describing the problem. Any `reply-with` parameter in the message causing the error should be used as the `:in-reply-to` parameter of the reply.

3.6 KQML Summary

To summarize, the main features of KQML as used in TRAINS-96 are:

- Regular, Lisp-like, easily-parsable syntax
- Reserved, common parameters
- Well-defined, intuitive performatives
- Content-independence

We have used this basis to build up a communication infrastructure for the TRAINS System that is simple to use and easy to extend.

4 Input Manager

Intermodule communication in TRAINS-96 is based on a star topology of modules exchanging KQML messages using socket-based inter-process connections. The TRAINS-96 Input Manager (IM) sits at the hub of this network, where it functions primarily as a router of KQML messages between other modules. The Input Manager is implemented as a standard Unix server process, listening for new connections on a "well-known" socket address and creating a new client context for each accepted connection. It then reads KQML messages from its client connections and either acts on them (if they were sent to the IM itself) or arranges for their delivery to the specified recipients. Clients are free to enter and leave the system at will. The Input Manager can arrange suitable notifications for modules that depend on the presence or status of other modules. The Input Manager also accepts and acts on messages addressed to it (IM). The complete set of these messages is shown in Table 5, and will be described in the rest of this section.

4.1 Client Registration

Once a module has established the socket connection to the Input Manager,[2] the first thing it must do is identify itself using the KQML `register` performative:

(`register :receiver im :name` *name*)

Multiple `register` messages are allowed, and it is assumed that whatever process is at the other end of the connection will sort out the recipients of messages sent down the pipe (if this matters). Typically, however, there is a one-to-one mapping of Input Manager connections to modules.

This registration is sufficient for the Input Manager to provide basic message routing using the `:receiver` parameter. As a convenience, if the `:sender` parameter is not given, and if the Input Manager knows the name of the sender (from a previous `register`), it fills in the `:sender` parameter before delivering the message. Note that we deliberately allow a module to specify another `:sender` in a message, since security is not an issue for TRAINS-96 while the ability to "forge" a message is essential for debugging and replay. Of course, this could be disabled if needed.

4.2 Selective Broadcast

Many modules in the TRAINS system act as broadcasters of information, for example a speech recognizer might broadcast the detection of a word, or the display might broadcast the occurrence of a mouse click. This is implemented as follows in TRAINS-96: if the `:receiver` parameter of a message received by the Input Manager is not specified, it is treated as a broadcast from the sender. The message is, however, only copied to modules that have previously registered interest in the sender's broadcasts. We call this "selective

[2]The manpage (Section D.6) provides details of how to do this, or see Section 5 concerning the Process Manager.

Performative	Content	Arguments
register	*n/a*	:name :class
request	listen	*name/class*
request	unlisten	*name/class*
request	define-class	*name* :parent
request	dump	
request	exit	*status*
request	chdir	*dirstring*
request	hide-window	
request	show-window	
tell*	start-conversation	:name :lang :sex
tell*	end-conversation	
tell	ready	
evaluate	status	*name/class*
monitor	status	*name/class*
unmonitor	status	*name/class*
broadcast	*performative*	

Table 5: Input Manager (IM) Messages (* indicates message is ignored)

broadcast," as opposed to a broadcast that is sent to all other modules unilaterally (which is also available, see the **broadcast** performative). The use of selective broadcast avoids modules needing to "ignore" broadcasts that they would otherwise receive, and reduces the amount of data transferred due to broadcast messages.

In order to register interest in a module M's broadcasts, a module S sends the following request to the Input Manager:

```
(request :receiver IM :content (listen M))
```

There is a corresponding **unlisten** request to stop receiving broadcasts. After this message has been received by the IM, if M sends a message like:

```
(tell :content (word "hello"))
```

then S (and any other listeners to M) will receive a message of the form:

```
(tell :content (word "hello") :sender M :receiver S)
```

The Input Manager fills in the **:sender** field (since it wasn't specified in the original message) and also fills in the **:receiver** field before sending the message. Note that this makes it impossible to distinguish between a broadcast and a direct send of a message if the contents are the same. We adopted this approach because it makes life easier for modules that manage connections for other modules, and in practice we have not had a need to distinguish the two cases.

4.3 Non-selective Broadcast

The selective broadcast technique is most useful for modules viewed as data generators, especially in conjunction with the use of module classes, described below. In other circumstances a true broadcast is necessary, where a message is sent to all connected modules. For example, in TRAINS-96, the `start-conversation` message is broadcast to all modules so that they can take appropriate action at the start of a session.

To send a non-selective broadcast, the sender must send a `broadcast` message to the Input Manager. The content of the message is the performative to send to all connected modules. For example:

```
(broadcast :receiver im :content (tell :content (start-conversation)))
```

If the Input Manager receives such a message from client `S`, then all connected clients `M` *except for* `S` will receive a message:

```
(tell :content (start-conversation) :sender S :receiver M)
```

The message is also processed by the Input Manager itself.

The key difference between selective and non-selective broadcast is that a module only receives selectively broadcast messages if it has asked for them (with a `listen` request). This is not true for non-selective broadcasts, which can therefore result in error replies (or worse!) if a module receives a message that it is not expecting and cannot understand. In TRAINS-96, we try to ensure that all modules at least accept those messages broadcast non-selectively, even if they don't do anything with them.

4.4 Module Status

The Input Manager also provides another important service as the hub of the TRAINS-96 communication architecture. It accepts and coordinates indications of module status, allowing modules to synchronize with each other.

To report that it is ready to receive messages, a module sends a message of the form:

```
(tell :receiver IM :content (ready))
```

Note that a module may receive messages before it has indicated that it is ready (indeed, this may be essential if it needs to make various queries before it is itself ready).[3] If a module `S` is interested in the status of a module `M`, it sends a request to the Input Manager:

```
(monitor :receiver IM :content (status M))
```

Immediately, and thereafter whenever the module `M`'s status changes, `S` will receive a message of the form:

```
(reply :sender IM :receiver S :content (status M X))
```

DEAD	Module has never connected
CONNECTED	Module is connected (and registered) but not ready
READY	Module has sent ready message
EOF	Module has closed connection

Table 6: Input Manager Status Indicators

where X is one of the status indicators listed in Table 6. A more complicated status-reporting scheme could be devised if needed, but this has proven sufficient for TRAINS-96. The evaluate performative can be used if only a single status report is needed, rather than continual monitoring.

4.5 Module Classes

An important improvement in TRAINS-96 is the addition of module classes to the Input Manager. Using this facility, modules can be grouped into classes, and these classes can be used as arguments in requests such as listen and monitor. This allows a module, for example, to listen for selective broadcasts from a class of modules, without knowing which modules exactly (by name) make up that class.

To specify the class to which a module belongs, an additional argument must be specified in the register message, as in:

(register :receiver IM :name *name* :class *class*)

The value of the :class parameter must be a known class (created with the define-class request), or a list of such classes. In TRAINS-96, we define the very simple hierarchy shown in Figure 3. The main use of these classes is to allow modules interested in user input (for example, the Parser, Section 12) to receive the selective broadcasts from that class.

4.6 Input Manager Display

The Input Manager also provides an X Window System display of module status and message traffic. A sample screenshot of the IM display is shown in Figure 4, showing the modules making up the TRAINS-96 system, including several implemented within the Discourse Manager (Section 13) using multiple register messages. This display was originally implemented as a separate module, which had the advantage of not cluttering the basic functions of the Input Manager. Unfortunately, this required that every message be copied to the display module, effectively doubling the message traffic. The Input Manager can be compiled without the display, should the overhead prove significant (it hasn't so far).

[3] A module must be registered in order to receive messages, of course, but it needn't be ready.

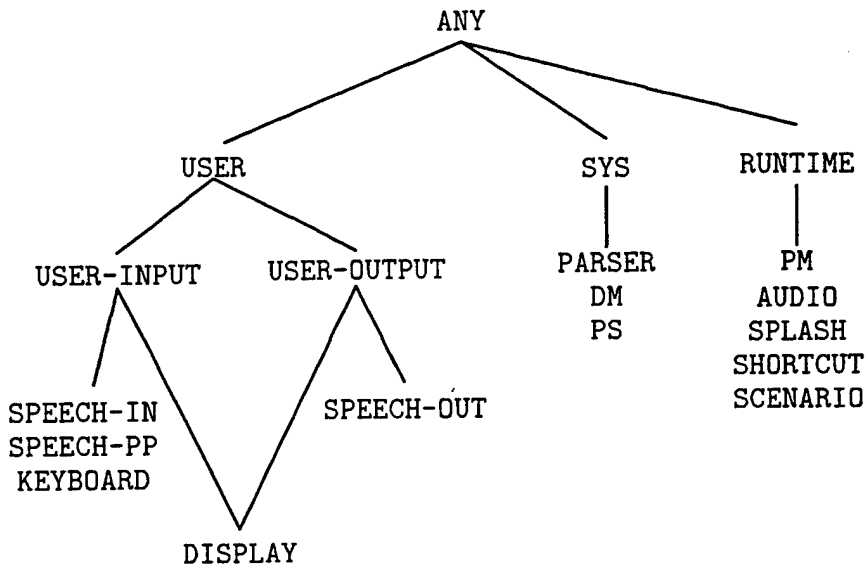

Figure 3: Input Manager Classes in TRAINS-96

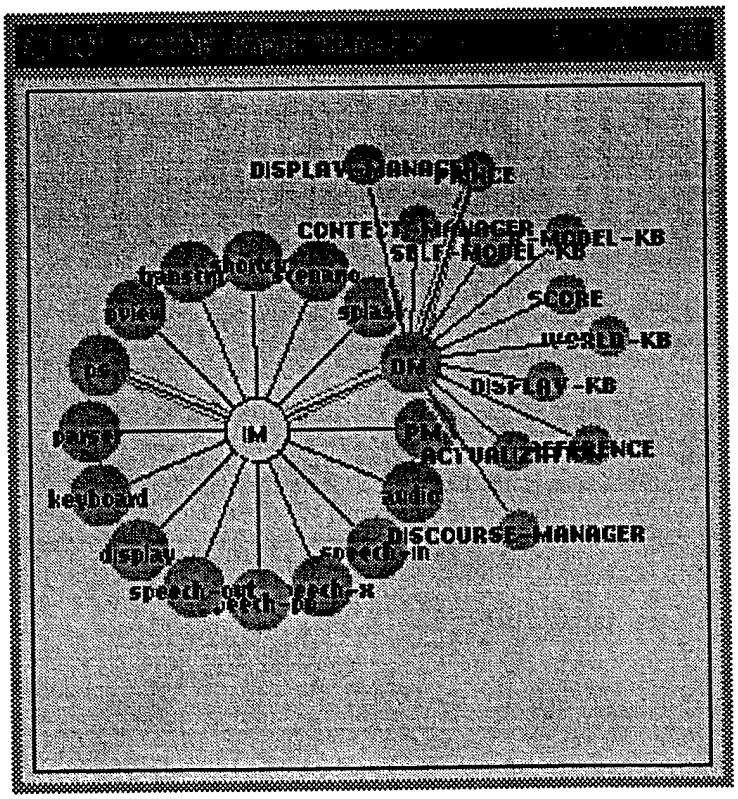

Figure 4: Input Manager Display (module status is indicated using color)

5 Process Manager

The Input Manager looks after inter-module communication once modules are connected to it. The other core requirement for a system like TRAINS-96 is therefore a means of launching and managing the processes that implement the modules. In earlier versions of the TRAINS system, both message passing and process management were provided by a single, complicated program. In TRAINS-96, we have split this into two programs, with the Process Manager handling the process management aspect, thereby simplifying both modules.

The main role of the Process Manager is to launch new processes and provide a means for signaling and killing them, based on KQML messages it receives via the Input Manager. The complete list of Process Manager messages is shown in Table 7.

5.1 Starting Processes

The `start` message requests that the Process Manager launch and manage a new Unix process. The `:name` parameter identifies the process for subsequent `kill` messages. The process is run on the host given by the `:host` parameter, or on the current host (*i.e.*, the same host as the PM itself is running on) if no `:host` is given. The executable file actually launched is given by the `:exec` parameter, which should be a complete pathname (*i.e.*, starting with a slash).

The argument list and environment of the newly-started process can be specified using the `:argv` and `:envp` parameters. The argument list, if given, should be a KQML list of tokens or strings. The first element of this list will be `argv[0]` (traditionally the name of the program), the second will be `argv[1]`, and so on. The environment list, if given, should be a KQML list of tokens or strings of the form "*var=value*", meaning that environment variable *var* has the given *value*. If no `:envp` is given, the process inherits the current environment. If `:envp` is given, the process' environment contains *only* those variables.

5.2 Input Manager Connection

As described in the previous section, the first thing a module in the TRAINS-96 System must do is connect to the Input Manager and send the `register` message to identify the connection. To simplify this procedure, the Process Manager will, by default, open a socket connection to the Input Manager and then launch the new process with its standard input and standard output connected to the IM.[4] It will use the `:name` parameter (and `:class` if given) to send an appropriate `register` message to the Input Manager before starting the process. The `:connect` parameter to the `start` request can be set to `nil` if this is not desired.

This is an extremely useful piece of functionality. It allows modules to be developed as simple Unix programs that read standard input and write standard output, without having to worry about socket programming and communication. This speeds development, aids

[4] This is similar to the role of `inetd(8)` in a Unix system.

Performative	Content	Arguments
`request`	`start`	`:name :class :host :exec` `:argv :envp :connect`
`request`	`kill`	*name* `:signal`
`request`	`dump`	
`request`	`exit`	*status*
`request*`	`chdir`	*dirstring*
`request*`	`hide-window`	
`request*`	`show-window`	
`tell*`	`start-conversation`	`:name :lang :sex`
`tell*`	`end-conversation`	

Table 7: Process Manager (PM) Messages (* indicates message is ignored)

debugging, and allows the modules to be used in novel ways by hooking their inputs and outputs together, for example using Unix shell pipelines in place of the Input Manager.

5.3 Remote Processes

The TRAINS System is designed to run on any combination of machines that support standard Unix networking (sockets, `rsh`, *etc.*). As noted above, the `:host` parameter can be used to specify remote execution of a module. In the current implementation, this is achieved by creating a small shell script named `pm.start.`*module* (in the PM's current directory) that sets the environment appropriately and then launches the desired program (`:exec` parameter). This script is then run remotely using the `rsh(1)` remote shell command.

There are two possible side-effects of this procedure. First, the Process Manager need to be run in a directory that is visible (and similarly named) to all machines on which modules will be run. For example, running it in `/tmp` will cause problems when the `/tmp` on the remote machine doesn't contain the script to be run. The second effect is that signals do not always propagate to remote processes, with the result that some remote processes may be left running after a process is killed (for example, when the Process Manager shuts down). We plan to solve both of these problems eventually, but for now at least you've been warned.

5.4 Other Process Manager Functions

The other Process Manager functions are trivial. The `kill` request sends the given signal (default `SIGTERM`, 15) to the named process. The `dump` request prints the Process Manager's internal data structures to standard error for debugging.

6 Audio Manager

One of the most difficult technical issues during the development of both TRAINS-95 and TRAINS-96 was the management of the workstation audio devices and resources for speech input and output, as well as other needs. Among the many problems were: (1) the speech recognizer (Section 7) and speech generator (Section 9) used different sampling rates, while our Sun workstation hardware supported only a single rate at once; and (2) we needed a reliable way to know when audio output was finished despite system buffering, in order to synchronize speech generation with other actions.

The TRAINS-95 system used two workstations, one for audio input and one for audio output, to work around these problems. This was clearly not the ultimate solution, so for early versions of TRAINS-96, we implemented an Audio Manager that could manage multiple client connections for both input and output at arbitrary sample rates. This module was controlled with KQML messages received via the Input Manager, although it then established "sideband" socket connections for the transfer of the audio data itself. This had the advantage that audio connections were recorded as part of the Input Manager KQML message traffic, which was useful during debugging of the audio clients. The Audio Manager was eventually undone by point (2) above, when synchronization issues with the Sun audio hardware became too difficult to do properly.

For the final version of the TRAINS-96 system, we are using the AudioFile network audio system developed at DEC [Levergood *et al.*, 1993a; Levergood *et al.*, 1993b]. This package is based on the X Window System sources and attempts to do for audio what X does for graphics, namely provide a network-transparent audio model based on a client-server architecture. The package has support for a variety of platforms, including Sun and DEC workstations, and could be ported to many others by using the X ports as a guide. The AudioFile server is started when TRAINS-96 is launched. Those modules that use audio connect to the server and transfer audio data using AudioFile library (`Alib`) routines.

The final TRAINS-96 Audio Manager is therefore implemented as an AudioFile client, and provides an X/Motif interface to audio input and output settings. It provides input and output level controls, a "VU" meter for adjusting record levels, and menus for switching between microphone and line inputs, and speaker, headphone and line outputs. The Audio Manager understands the core set of messages shown in Table 3 (page 9), although it ignores them all with the exception of `exit` and `hide/show-window`.

7 Speech Recognition

Speech recognition in TRAINS-96 is performed by the SPEECH-IN module, which is based on the Sphinx-II system from Carnegie Mellon University [Huang *et al.*, 1992]. We made as few changes as possible to the Sphinx code, mostly adding KQML input and output, fixing a few bugs, and adjusting the audio routines to use the AudioFile server as described in the previous section. A more substantial change involved enhancing the program's "live" mode to provide incremental ssegmentation and scoring. The result is a "faceless" speech recognition engine controlled by KQML messages with incremental output also in the form of KQML messages. The complete set of messages understood by the SPEECH-IN module is shown in Table 8.

To keep the speech recognition module as simple and as flexible as possible, it has no user interaction components (display, controls, *etc.*) of its own. Instead, once it receives a `start` message, it will start processing audio and outputting recognized words until a `stop` message is received. In TRAINS-96, the `start` and `stop` messages are generated by the Speech Controller module (SPEECHX, Section 15.1), among others.

7.1 Speech Recognizer Operation

Recognition results are output incrementally as the Sphinx-II engine settles on a hypothesis. All outputs are selective broadcasts (*i.e.*, there is no `:receiver` specified), and are sent to `LISTEN`ing modules by the Input Manager (see Section 4.2). Each utterance begins with a message of the form

`(tell :content (start :uttnum` *N*`))`

This is sent when SPEECH-IN receives a `start` message, and the utterance number N is incremented for each utterance in a conversation. The utterance counter is reset when SPEECH-IN receives a `start-conversation` message.

When the Sphinx-II engine indicates that its hypothesis has changed, the new hypothesis is compared with what has already been broadcast and new messages are sent. In the normal case, the hypothesis has been extended with a new word, and the following message is sent:

`(tell :content (word "`*string*`" :uttnum` *N* `:index` *I* `:frame` *F* `:score` *S*`))`

The text of the word is formatted as a KQML string in double quotes. The index I refers to "index positions," the gaps between words as understood by the Parser. Index position 1 precedes the first word of the utterance. When the recognized word is actually several words (such as "`I_WANT`" or "`LET'S`"), the index I is a list consisting of the starting and ending indices. The frame parameter F is a list of two numbers indicating the range of acoustic frames in the Sphinx-II audio input spanned by the word. The score S is an estimate of the accuracy of the recognition at that word, based on Sphinx-II's acoustic score normalized by the number of frames.

In addition to generating new words that extend the hypothesis, Sphinx-II can also revise a previous part of the hypothesis in light of further recognition. For example, "`SINCE`

Performative	Content	Arguments
request	start	
request	stop	
request	exit	*status*
request	chdir	*dirstring*
request*	hide-window	
request*	show-window	
tell*	start-conversation	:name :lang :sex
tell*	end-conversation	

Table 8: Speech Recognizer (SPEECH-IN) Messages (* indicates message is ignored)

IN AT" may get revised to "CINCINNATI" when the final syllable is processed. In this case, the SPEECH-IN module sends the following message:

(tell :content (backto :uttnum *N* :index *I*))

This message invalidates words at all indices greater than or equal to *I*. When *I* is 1, the entire hypothesis is invalidated. Since Sphinx-II often changes its mind about words on the frontier of the recognition, we have implemented a form of buffering to reduce the number of backto messages generated by SPEECH-IN. With this, the last k words of the hypothesis are not output (until the utterance is complete). The value of k is controllable using a command-line argument, and defaults to 2.

Finally, when SPEECH-IN receives a stop message, it stops processing audio and causes Sphinx-II to terminate its search and output a final hypothesis. Appropriate word and/or backto messages are generated, and when Sphinx-II indicates that the search is complete, the following message is sent:

(tell :content (end :uttnum *N*))

It is at this point that further processing of the utterance, *e.g.*, by the Parser, usually begins.

7.2 Speech Recognizer Files

The SPEECH-IN module also records information in files stored in the directory given by the last chdir message it received (or the current directory if no chdir messages have been received). These include a file "sphinx.log" containing initialization parameters and messages from Sphinx-II and, for each utterance, files "utt.*N*.au" and "utt.*N*.out", containing the audio data and Sphinx-II output, respectively, for utterance *N*. The audio datafiles consist of signed 16-bit linear PCM-encoded samples at 16000 Hz with no audio headers. The logs contain a summary of the search results and the final hypothesis.

8 Speech Post-Processor

8.1 Description

The TRAINS-96 system includes a Speech Post-Processor module (SPEECH-PP) for correcting the word-by-word messages coming from SPEECH-IN, the speech recognition module. The post-processor listens to the message stream from the speech recognizer and provides its own parallel output for use by the parser. In short, the purpose of the post-processor is to improve word recognition accuracy and to thereby increase the likelihood of a successful interchange between the user and the system. Further discussion of the post-processor's design can be found in [Ringger, 1995], and a detailed experimental analysis of its performance can be found in [Ringger and Allen, 1996].

The complete set of messages understood by the SPEECH-PP module is shown in Table 9.

8.2 Speech Post-Processor Models

The SPEECH-PP module requires two models to perform helpful corrections: a language model and a channel model. It is necessary to train these models prior to operation. The training process will be described in a future report. Briefly, the language model describes the likelihood of word collocations in the language a typical user will employ during a session with the TRAINS-96 system. The channel model, which consists of several components, reflects common erroneous behavior of the SPEECH-IN module,[5] in terms of word-level confusions.

For TRAINS-96, two sets of models are available: one for ATIS (distributed with Sphinx-II) and one for TDC-75 (developed from sessions with the TRAINS-95 system as well as the TRAINS-93 Dialogue Corpus [Heeman and Allen, 1995]). Each set must only be used when SPEECH-IN is using its language model with the corresponding name. Using the wrong SPEECH-PP model will defeat the port-processor's purpose entirely. For TRAINS-96, the TDC-75 models are the default for both SPEECH-IN and SPEECH-PP. Their components are listed in Table 10.

These models are employed in the post-processor's Viterbi search. For a given input word sequence, the search considers and scores all possible corrections licensed by these models (within some score threshold, or "beam"). The scores for each correction hypothesis are derived exclusively from the model files.

8.3 Speech Post-Processor Operation

As with the SPEECH-IN module, the speech post-processor has no user interface components (display, controls, *etc.*) of its own. Instead, once it receives a `start` message from

[5] Although it has only been trained for and used with the Sphinx-II speech recognition system from Carnegie Mellon University, this technique can be trained for and used with other continuous speech recognizers.

Performative	Content	Arguments
`tell`	`word`	*string* `:uttnum :index :frame`
`request`	`start`	`:uttnum`
`request`	`end`	`:uttnum`
`request`	`offline`	`t|nil`
`request`	`exit`	*status*
`request`	`chdir`	*dirstring*
`tell*`	`start-conversation`	`:name :lang :sex`
`tell*`	`end-conversation`	

Table 9: Speech Post-Processor (SPEECH-PP) Messages (* indicates message is ignored)

File-name	Description
`tdc-75.arpabo`	ARPA-style Back-off language model
`tdc-75.wbic`	Word bigram count table (non back-off lang. model)
`tdc-75.confusion`	$1 \rightarrow 1$ channel model
`tdc-75.inserted`	$0 \rightarrow 1$ channel model (currently unused)
`tdc-75.mergecand`	$1 \rightarrow 2$ channel model
`tdc-75.splitcand`	$2 \rightarrow 1$ channel model

Table 10: Speech Post-Processor Models

SPEECH-IN, it will start processing the `word` messages from SPEECH-IN and sending corrected words until an `end` message is received. Corrected words are sent incrementally as the post-processor settles on the most likely correction hypothesis. All outputs are selective broadcasts (*i.e.*, there is no `:receiver` specified), and are thus sent to `LISTEN`ing modules by the Input Manager. Each turn begins with a message of the form

`(tell :content (start :uttnum` N `))`

This is sent when SPEECH-PP receives a `start` message from SPEECH-IN, and the turn (or utterance) number N is copied from the SPEECH-IN message.

When the post-processor's hypothesis changes, the new hypothesis is compared with the whole previously broadcast hypothesis, and new messages are sent. In the typical case, the hypothesis has simply been extended with a new word, and the following message is sent accordingly:

`(tell :content (word "`*string*`" :uttnum` N `:index` I `:score` S `))`

The text of the word is formatted as a KQML string in double quotes. The index I refers to index positions, the gaps between words as understood by the parser.[6] For a more detailed

[6]The next version of the post-processor will use acoustic frame numbers in place of index positions.

discussion of index positions, see Section 7 regarding Speech Recognition and the SPEECH-IN module. The score S is a probability estimate of the accuracy of the correction for that word, based on both models.

In addition to generating new words that extend the hypothesis, the post-processor can also revise a part of the hypothesis already broadcast, in light of subsequent input messages. For example, "TO" may be replaced by "TOLEDO" when "LEAVE" appears from the recognizer. In this case, the SPEECH-PP module sends the following message:

```
(tell :content (backto :uttnum N :index I ))
```

This message invalidates words at all indices greater than or equal to I. When I is 1, the entire hypothesis is invalidated. Since the post-processor often changes its mind about words on the frontier of the recognition, it reduces the number of necessary `backto` messages by buffering one word position. Providing `backto` notification is complicated by the fact that `backto` messages also appear on the input as they come from SPEECH-IN.

Finally, when SPEECH-PP receives an `end` message, it stops expecting incoming messages, terminates its search, and outputs its final hypothesis, including the contents of the frontier buffer, by sending appropriate `word` and/or `backto` messages. Once the final word messages have been sent, the following message is sent:

```
(tell :content (end :uttnum N ))
```

The SPEECH-PP module also records information in a log file called `speechpp.log`, stored in the directory given by the last `chdir` message received (or the current directory if no `chdir` messages have been received). The log file begins with a listing of configuration parameters and contains a record of all incoming and outgoing message traffic along with a record of the complete hypothesis at any stage in a given turn.

Lastly, the `offline` message is sent by the Speech Controller module (SPEECH-X, 15.1) in response to user control. An argument of `t` causes the Post-Processor to go off-line. The Parser then receives only the messages sent by SPEECH-IN. An `offline` message with an argument of `nil` brings the Post-Processor back online.

9 Speech Generation

Speech generation in TRAINS-96 is performed by the SPEECH-OUT module, which acts as an interface to the TrueTalk generation engine from Entropics, Inc. A diagram of the relationship between SPEECH-OUT, the TrueTalk server, and the AudioFile audio server is shown in Figure 5. At startup, connections are established to the two servers, and generation is then controlled by KQML messages sent to SPEECH-OUT. The complete set of messages understood by SPEECH-OUT is shown in Table 11.

As is obvious from the table, the SPEECH-OUT module has only one purpose. Upon receipt of a **say** request, it passes the string to the TrueTalk engine for synthesis. The resulting audio data is returned to SPEECH-OUT through the TrueTalk API, and is then sent to the AudioFile server for output. When the audio server indicates the audio output is complete, SPEECH-OUT generates the following reply:

```
(reply :content (done))
```

Of course, any `reply-with` in the original **say** request is used as the `in-reply-to` for the reply. This reply can be used to synchronize speech generation with other activities, such as displays.

One technical issue bears mentioning here. The TrueTalk server can synthesize speech at either 8 kHz or 12 kHz (signed 16-bit linear PCM samples). Due to limitations of the AudioFile server, and the fact that Sphinx-II requires 16 kHz samples for recognition, we use the 8 kHz generation rate from TrueTalk and convert it to 16 kHz by linear interpolation. The AudioFile server provides smooth playback and the important ability to determine when audio playback has completed.

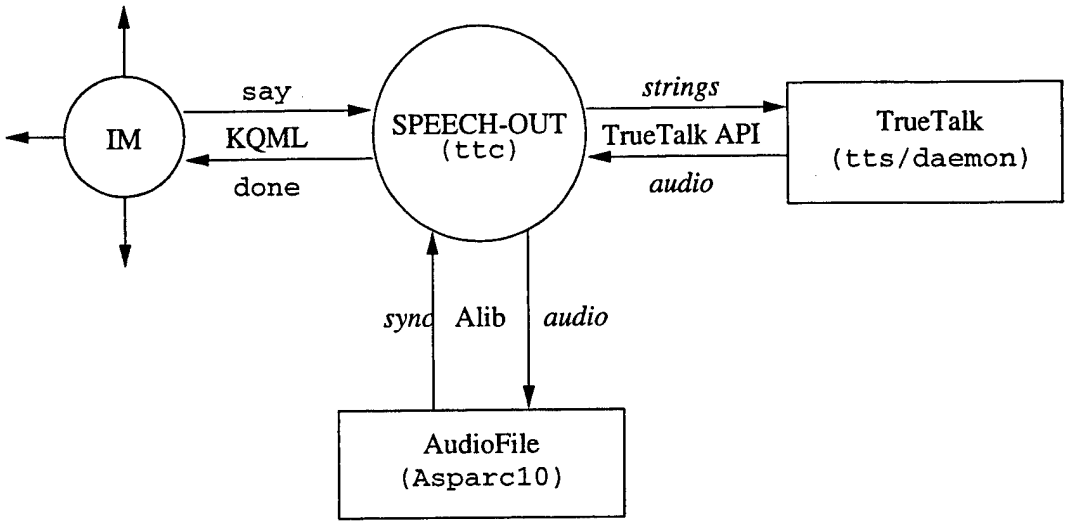

Figure 5: SPEECH-OUT Functional Diagram

Performative	Content	Arguments
`request`	`say`	*string*
`request`	`exit`	*status*
`request*`	`chdir`	*dirstring*
`request*`	`hide-window`	
`request*`	`show-window`	
`tell*`	`start-conversation`	`:name :lang :sex`
`tell*`	`end-conversation`	

Table 11: Speech Generator (SPEECH-OUT) Messages (* indicates message is ignored)

10 Keyboard Manager

The TRAINS-96 Keyboard Manager module, KEYBOARD, is responsible for typed input and other keyboard-related issues. It understands the basic set of KQML messages shown in Table 3 (page 9), although it ignores `start-` and `end-conversation` as well as `chdir`.

The Keyboard Manager presents a window into which the user can type and incrementally outputs typed words as KQML messages of the form:

(tell :content (word "*string*" :index *I*))

The text of the word is formatted as a KQML string in double quotes. The index I refers to index positions, the gaps between words as understood by the parser. For a more detailed discussion of index positions, see Section 7 regarding Speech Recognition and the SPEECH-IN module. The Keyboard Manager buffers a small number of words internally to allow the user to make corrections.

When the user hits the Return key, the Keyboard Manager sends any buffered words and then sends a message of the form:

```
(tell :content (end))
```

When the user backspaces over a previously-output word, the Keyboard Manager sends a message of the form

(tell :content (backto :index *I*))

to invalidate words at all indices greater than or equal to I. When I is 1, the entire hypothesis is invalidated. These messages are identical to those generated by the speech processing modules (SPEECH-IN, Section 7, and SPEECH-PP, Section 8) except that the Keyboard Manager does not maintain any numbering of the user's utterances.

One technical issue is worth mentioning here. When the Keyboard Manager's window is visible, it places an active grab on the user's keyboard in order to receive keyboard input regardless of the location of the mouse pointer on the screen. A hotkey (currently the left "Alt" key) allows the user to send `start` and `stop` messages to the speech recognizer, allowing simultaneous use of the mouse with spoken input. The keyboard grab can be enabled and disabled using a command-line option or using the Keyboard Manager menus. For more details, see the manpage (Section D.11, page 101).

11 Display

The TRAINS-96 Display module provides an object-oriented map display that forms the main graphical interface of the TRAINS-96 system. A sample screenshot is shown in Figure 6. Map objects can be created, modified, and destroyed dynamically using KQML messages received via the Input Manager. In addition, KQML messages are generated by the Display in response to user actions such as mouse clicks and drags. The Display module is implemented as a standard X Window System client, which ensures portability across a large range of platforms.

The interface language used by the Display module is among the most complicated of any module in the TRAINS-96 System. The complete set of messages understood by the Display module is shown in Table 12. Due to the complexity of this interface, the messages will be described in several sections.

11.1 Object Manipulation Messages

The `create` message is the most important but also the most complicated message understood by the Display. The possible attributes are shown in Table 13. The `:name` and `:type` attributes are mandatory—most of the others have suitable defaults.

The `:name` attribute names the new object and can be used subsequently to refer to it. The `:type` attribute sets the type of the object, currently one of `city`, `track`, `engine`, `route`, or `region`. The `:displayed` attribute determines whether the object is visible (and selectable) or not. The `:depth` attribute sets the depth of the object on the display–deeper objects (greater `:depth`) are drawn under shallower ones. The `:bg` attribute indicates whether the object is considered part of the "background" (*i.e.*, is dynamic or not). See the `setbg` request, below, for details. The `:color` and `:fillcolor` attributes are X color names (or X RGB color specs). The `:fill` attribute varies from 0 (unfilled) to 100 (completely filled), and the `:thickness` attribute affects the borders of the object.

The shape of the object can be specified using the `:shape` attribute and one of the forms listed in the table, although most object types also have default shapes. The default shape for a track is a line connecting the endpoints, for an engine it is a simple schematic engine thingo, and for a route it is a series of spline curves along the tracks in the route. The *loc* attributes (locations), can be the names of objects, meaning their centers, or a list `(X Y)` of coordinates in the Display window (the origin is the upper, left corner). For polygon and multiline shapes, `:points` means the locations are absolute coordinates and `:rpoints` means they are relative (after the first one, of course).

Finally, the create request can specify attributes specific to the type of object being created. These additional attributes are shown in Table 14. The `:orientation` attribute can be one of `north`, `northeast`, `east`, *etc.*, describing the position of the label relative to the object (for cities) and the position of the engine relative to the city it is `:at` (for engines). The `:outlined` attribute for engines is an alternative for `:fill 0` or `:fill 100`.

The other Display object manipulation messages are pretty much self-explanatory. The `destroy` request destroys the named object, removing it from the Display. The `display` and

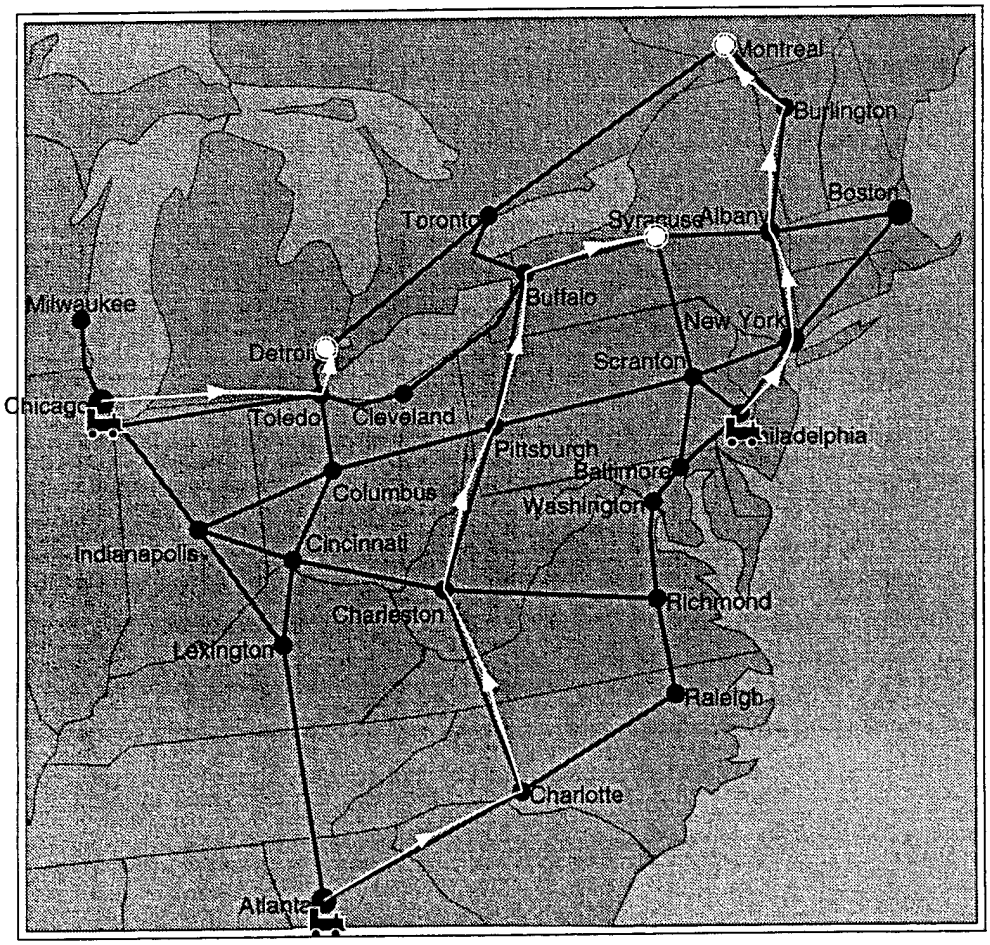

Figure 6: Sample DISPLAY module map display, showing regions, cities, tracks, engines, routes, and highlights

Performative	Content	Arguments
Object Manipulation		
request	create	:name :type ... *see Table 13*
request	destroy	*obj*
request	display	*obj*
request	undisplay	*obj*
request	set	*obj attr-value pairs*
request	default	*attr-value pairs*
Object Highlighting		
request	highlight	*obj* :color :type :flash
request	unhighlight	*obj* :color :type :flash
Dialog Boxes		
request	confirm	*tag str*
request	dialog	*type str*
Display Control		
request	canvas	:title :height :width
request	translate	X Y
request	scale	X Y
request	setbg	
request	say	*str*
request	postscript	*str*
request	refresh	
request	restart	
request	map	*str*
Common Messages		
request	exit	*status*
request*	chdir	*dirstring*
request	hide-window	
request	show-window	
tell	start-conversation	:name :lang :sex
tell	end-conversation	

Table 12: Display (DISPLAY) Messages (* indicates message is ignored)

Attribute	Value
:name	*string*
:type	city, track, engine, route, or region
:displayed	t|nil
:depth	*N*
:bg	t|nil
:color	*string*
:fillcolor	*string*
:fill	*N*
:thickness	*N*
:shape	(circle :center *loc* :radius *N*)
	(poylgon :center *loc* :points|:rpoints (*loc1* *loc2* ...))
	(multiline :points|:rpoints (*loc1* *loc2* ...))

Table 13: Display Object Creation Attributes

Type	Attribute	Value
city	:label	*str*
	:orientation	*orientation*
	:ptsize	*N*
track	:start	*loc*
	:end	*loc*
engine	:at	*loc*
	:orientation	*orientation*
	:outlined	t|nil
route	:start	*city*
	:tracks	(*obj1* *obj2* ...)

Table 14: Display Object-specific Creation Attributes

`undisplay` requests toggle whether an object is visible (and selectable). The `set` request allows attributes of the object to be changed, and the `default` request allows the default values of attributes for future `creates` to be set.

11.2 Object Highlighting Messages

The `highlight` request causes the given object to be highlighted. The Display supports both `object` highlights, where the color of the object is changed, and `circle` or `rectangle` highlights where the appropriate shape is drawn around the object. The `:flash` attribute can be `nil` meaning don't flash (the default), `t`, meaning flash the highlight forever, or a number, meaning that the highlight should flash that many times and then unhighlight. Multiple highlights can be applied to an object and they are rendered in the order they were applied. The `unhighlight` request removes the matching highlight from the given object. If no attributes are given, all highlights are removed from the object.

11.3 Dialog Box Messages

The Display module supports two types of dialog box. A blocking confirmer is presented in response to a `confirm` request. When the user selects either "OK" or "Cancel," the Display sends a reply indicating the choice. The `dialog` request, on the other hand, puts up a non-blocking dialog box containing some information, for example the goals of the current scenario. This is obviously something that will need to be developed as we start using more complicated multi-modal displays.

11.4 Display Control Messages

These messages either affect the internal operation of the Display module or perform somewhat auxiliary functions. The `canvas` request sets the title, height and width of the Display window. The `translate` and `scale` requests set global translation and scaling factors that are used for subsequent `create` requests. The `setbg` request sets the background pixmap of the Display's window to include any objects whose `:bg` attribute is set to `T`. These objects are then not redrawn during Display updates. This is typically used once per map after the map objects have been created but before any engines, routes, *etc.*, have been created. It is necessary to minimize flashing during redraws of complicated displays. The `say` request adds text to the "system output" window above the map display, the `postscript` request dumps a Postscript file describing the current display, the `refresh` request redraws the display, and the `restart` request destroys all objects and rereads the initial mapfile. Finally, as indicated previously, the mapfiles are simply files containing these very same messages—the `map` request causes the Display Module to read the indicated mapfile.

11.5 Display Output Messages

The Display module also generates messages in response to user mouse and keyboard actions, although the latter function has now been taken over by the Keyboard Manager

Performative	Content	Arguments
`tell`	`mouse`	`:select` *obj1 obj2* ...
`tell`	`mouse`	`:drag` *obj* `:from` *obj* `:to` *obj1 obj2* ...
`tell`	`word`	*word* `:index`
`tell`	`backto`	`:index`
`tell`	`end`	

Table 15: Display Output Messages

(Section 10). These messages are summarized in Table 15. The `mouse` messages provide a list of objects "close" to the location of the mouse click, relying on other modules to do whatever context-dependent disambiguation might be necessary (*e.g.*, the system might be expecting a city). The other messages use the same protocol for buffered, incremental output of the user's typed input as the speech recognition modules SPEECH-IN (Section 7) and SPEECH-PP (Section 8).

12 Parser

One of the main intellectual claims embodied in TRAINS-96 (and in the TRAINS Project in general) is that user actions and utterances should be interpreted as *linguistic* actions, that is, as language. For this reason, all user input, including words from speech recognition, typed input, and graphical gestures, are interpreted by the Parser module. From this input, the Parser generates a logical form that serves as the first stage of interpretation of the user's "utterance" (where the quotes emphasize that we are not concerned solely with spoken language). This section provides a general introduction to the Parser and how it fits into the System, followed by a detailed description of the messages it understands and of the logical forms that it generates. The aim is to provide both an understanding of the current state of the Parser as well as the issues involved in extending it beyond the TRAINS-96 domain and problems. Since the Parser is the first interpretive stage of the TRAINS-96 System, this will also amount to a description of the linguistic capabilities of the System as a whole.

12.1 Parser Implementation

The Parser is implemented as a Common Lisp image. It reads KQML messages from standard input (Lisp stream `*standard-input*`) and writes KQML messages to standard output (Lisp stream `*standard-output*`).

12.2 Parser Input

The parser can receive words one at a time, or in groups. It is designed to receive messages directly from the speech recognizer (sender SPEECH-IN) and the post-processor (sender SPEECH-PP). If it receives post-processor messages for an utterance, it ignores the speech recognition messages. This way, if the post-processor is disabled or offline, the system still runs. The messages are the same format from either source, and are documented in Sections 7 and 8.

Typed words from the Keyboard module (see Section 10) are interpreted identically to words recognized by the speech modules. There is one difference worth noting: the Keyboard module provides neither utterance numbers nor a `start` message. The Parser handles the former by ignoring the utterance number and the latter by interpreting a `word` following an `end` as an implicit `start`.

The Parser also understands messages indicating user mouse actions sent by the Display module (see Section 11). The exact interpretation of these messages is still somewhat *ad hoc* and better handling of them is the subject of current research.

12.3 Parser Operation

Given that errors are inevitable, robust parsing techniques are essential. The Parser module uses a pure bottom-up parser based on that described in [Allen, 1995] to identify the

possible constituents at any point in the utterance based on syntactic and semantic restrictions. Every constituent in each grammar rule specifies both a syntactic category and a semantic category, plus other features to encode co-occurance restrictions as found in many grammars. The semantic features encode selectional restrictions, most of which are domain-independent. For example, there is no general rule for PP attachment in the grammar. Rather there are rules for temporal adverbial modification (*e.g.*, "at eight o'clock"), locational modification (*e.g.*, "in Chicago"), and so on. Documentation on the form of the grammar and the functions of the parser are found in [Allen, 1995].

The end result of parsing is a sequence of speech acts rather than a syntactic sentence. Viewing the output as a sequence of speech acts has significant impact on the form and style of the grammar. It forces an emphasis on encoding semantic and pragmatic features in the grammar. There are, for instance, numerous rules that encode specific conventional speech acts (*e.g.*, "That's good" is a CONFIRM, "Okay" is a CONFIRM/ACKNOWLEDGE, "Let's go to Chicago" is a SUGGEST, and so on). Simply classifying such utterances as sentences would miss the point. Thus the parser computes a set of plausible speech act interpretation based on the surface form, similar to the model described in [Hinkelman and Allen, 1989].

We use a hierarchy of speech acts, shown in Table 16, that encode different levels of vagueness, including an act (SPEECH-ACT) that indicates content without an identified illocutionary force. This allows us to always have an illocutionary force identified, which can be refined as more of the utterance is processed. The final interpretation of an utterance is the sequence of speech acts that provides the "minimal covering" of the input, *i.e.*, the shortest sequence that accounts for the input. The speech act forms reflect the surface form of the utterances, and fall into classes, organized along the lines of Searle's taxonomy [Searle, 1969]. Note that there are some acts, such as WARN, that do not appear as output from the parser but may be recognized by the system at later stages of processing. This is because there are no surface indicators of warning that are relevant to this domain (in another domain the utterance "Watch out!" might be relevant and be interpreted as a warning). If an utterance were completely uninterpretable, the parser would still produce an output - namely, a TELL act with no identified content! The full specification of the output of the parser can be found in [Allen, 1996a].

For example, consider a garbled utterance from an actual TRAINS-96 session:

Okay now I take the last train in go from Albany to is

The best sequence of speech acts covering this input consists of three acts:

1. a CONFIRM ("Okay");
2. a TELL, with content to take the last train ("Now I take the last train");
3. a REQUEST to go from Albany ("go from Albany").

Note that the "to is" at the end of the utterance is simply ignored as it is uninterpretable. While not present in the output, the presence of unaccounted words will lower the confidence score assigned by the parser to the interpretation.

The actual utterance was:

Representative Acts	
TELL	The most generic act, used for simple declarative sentences and for speech acts that were constructed from fragments with no clue as to the speech act.
ID-GOAL	A declarative utterance that explicitly introduces a goal, such as "I want to go to Boston."
Directive Acts	
REQUEST	The generic directive act. Used for imperatives and for fragmented utterances with clue words such as "please."
SUGGEST	For utterances that syntactically capture suggestions such as "Let's do X," and "How about X?"
YN-QUESTION	For interrogative sentences, and eventually for other utterances with question clues (such as intonation).
WH-QUESTION	For sentences involving WH queries, and fragments involving WH terms.
Expressive Acts	
EVALUATION	For utterances that express an evaluation such as "Good," "Bad," "That's good," *etc.*
APOLOGIZE	Conventional utterances such as "I'm sorry."
NOLO-PROBLEMO	Convention response to APOLOGIZE such as "No problem," ...
Other Conventional Acts (Declaratives?)	
GREET	Conventional greetings such as "Hi," "Hello," ...
CLOSE	Conventional closings such as "Bye," "See ya," ...
REJECT	Utterances that reject a proposal such as "No," and utterances with "instead," such as "Go through Avon instead of Bath."
CONFIRM	Utterances that confirm a proposal or answer a yes/no question, such as "Yes."
ACKNOWLEDGE	Utterances that may simply be acknowledging the interaction or confirmed, such as "OK," "uh huh," ...
NOLO-COMPRENDEZ	A small class of acts indicating non-comprehension, such as "huh?" and "What?"

Table 16: Parser Speech-Act Types

Okay now let's take the last train and go from Albany to Milwaukee.

Note that while the parser is not able to reconstruct the complete intentions of the user, it has extracted enough to continue the dialogue in a reasonable fashion by invoking a clarification subdialogue. Specifically, it has correctly recognized the confirmation of the previous exchange (1), and recognized a request to move a train from Albany (3). Act 2 is an incorrect analysis, and results in the system generating a clarification question that the user in fact ended up ignoring (in the actual session). Thus, in terms of the goal of furthering the conversation, the analysis produced by the robust Parser is fairly accurate.

12.4 Parser Output

For each user utterance, the Parser generates a `tell` performative (with no `:receiver`, *i.e.*, selectively broadcast) whose `:content` is a logical form describing the utterance's content. In this logical form, information about the sentence is captured in a set of slots. For sentences that parse completely, this can be viewed merely as an implementation issue. For ill-formed sentences, however, this representation allows the parser to produce partial interpretations using the same format as for fully-parsed sentences. This makes it easier to develop robust processing mechanisms in the later stages of processing.

Consider an example. The output for the sentence "Go from Avon to Bath, please" is shown as Figure 7. As you can see, the output is a list consisting of the following:

- The speech act performed. In this case, `SA-REQUEST`.
- Objects mentioned in the utterance. In this case, these are the logical forms for the noun phrases "Avon" and "Bath".
- Paths mentioned. In this case, this is the logical form of "from Avon to Bath".
- Semantics, which captures the full propositional content of the utterance. In this case, it is the action of going from Avon to Bath. Note the `LSUBJ` of the proposition is the special form `*YOU*` indicating the "understood" subject of the imperative.
- Noise: words that were unknown to the parser (none in this example).
- Social-context: indication of politeness and other factors. In this case, "please" is represented by `(POLITE PLEASE)`.
- Reliability: a score between 0 and 100 indicating the confidence the parser has in this result. This is most useful when the parser only partially parsed the utterance. Generally, scores above 75 are pretty good, while scores below 25 should be taken with a grain of salt.
- Mode: whether the input was typed, spoken, or graphical (useful for generating responses).
- Syntax: identifies parts of the logical form associated with important syntactic constructs, such as the subject, object, *etc.*

```
(SA-REQUEST
  :OBJECTS ((DESCRIPTION (STATUS NAME) (VAR V2242)
             (CLASS CITY) (LEX AVON) (SORT INDIVIDUAL))
            (DESCRIPTION (STATUS NAME) (VAR V2253)
             (CLASS CITY) (LEX BATH) (SORT INDIVIDUAL)))
  :PATHS ((PATH (VAR V2238) (CONSTRAINT
                             (AND (FROM V2238 V2242)
                                  (TO V2238 V2253)))))
  :SEMANTICS (PROP (VAR V2231) (CLASS GO-BY-PATH)
              (CONSTRAINT (AND (LSUBJ V2231 *YOU*)
                               (LCOMP V2231 V2238))))
  :NOISE NIL
  :SOCIAL-CONTEXT (POLITE PLEASE)
  :RELIABILITY 100
  :MODE TEXT
  :SYNTAX ((SUBJECT . *YOU*) (OBJECT))
  :SETTING NIL
  :INPUT (GO FROM AVON TO BATH PLEASE))
```

Figure 7: Sample Parser Output

- Setting: identifies temporal and location information in the utterance.
- Input: the words associated with this particular interpretation (since in general the input may be accounted for by several interpretations).

Further details of the Parser's logical form, including significantly more detail regarding the representation of the semantics, can be found in [Allen, 1996a].

13 Discourse Manager

The Discourse Manager is responsible for interpreting the user's utterances, and figuring out what to do about them. For problem-related utterances such as suggestions, this involves passing them on to the Problem Solving module and integrating the results. Before doing so, however, the Discourse Manager must first determine exactly what has been said by the user, fill in any elided or presumed information, and use the discourse state to check the salience of the current utterance (and response) to generate a preferred interpretation. The Discourse Manager is also responsible for generating system output using the DISPLAY and SPEECH-OUT components. The rest of this section provides a very high level overview of the Discourse Manager's operation.

13.1 Discourse Manager Implementation

The TRAINS-96 Discourse Manager is implemented as a Common Lisp image. Rather than using the standard input and standard output to exchange KQML messages via the Input Manager, the DM opens its own socket connection to the IM and communicates via that connection.[7] This means that if the DM is started by the Process Manager (PM), the `start` request should specify "`:connect nil`" to leave standard input and output attached to the terminal.

The Discourse Manager for TRAINS-96 is made up of a number of components, most of which are described further in this section:

Context Manager: Manages discourse contexts and performs operations such as pushing a new context, *etc.*

Reference: Resolves indexical and anaphoric references using discourse context

- **User Model:** Maintains system's view of what the user knows, intends, *etc.*
- **World Model:** Maintains system's view of the external world

Prince: Main "verbal reasoner," responsible for disambiguating speech acts, communicating with Problem Solver, and coordinating output.

Actualization: Turns system output needs into explicit requests for Display and/or Speech Generator modules.

- **Self Model:** Maintain's system's view of it's own beliefs and intentions
- **Display Model:** Maintains system's view of what is on the display, suitable for use in generation and in resolving indexical references (*e.g.*, "the red engine")

While the components are presented hierarchically, in fact the components are separate (within the same Lisp image) and communicate by exchanging KQML messages (internally)

[7] This was necessitated by a desire to run the DM using Allegro CL's Emacs interface during debugging although in the main system the DM runs as a separate Lisp image.

The dependency relation above shows which modules are closely tied, *e.g.*, that reference is the heaviest user of the World Model, not that no other module will use it. For example, Reference may access all of the Models and the Problem Solver. To resolve "the blue train" it needs to query the Display Model to find which train is being presented as blue, to resolve the "train in Cincinnati" it may need to ask the Problem Solver which train is involved in a plan to go through or end up in Cincinnati, and so on.

13.2 Discourse Manager Operation

The input to the DM is the speech act output by the Parser (see Section 12).

Discourse Context and Reference

This set of speech acts is first looked over by the Context Manager to store any lexical information that may influence later anaphor resolution and generation (*e.g.*, the utterance subject and object).

It is then passed to the Reference component, which has two duties. First, it resolves any noun references to their ground representations (that is, representations maintained by the World Model) if possible. Then, it does illocutionary remapping of the speech acts assigned by the parser as needed to fit discourse and reference cues. For instance, an utterance that consists of a naked REJECT "no" followed by a REQUEST "go via syracuse" will have the latter REQUEST remapped into the REJECT; it is essentially the content of the REJECT, not a separate REQUEST. After this processing, Reference returns the speech act(s), now in an internal format, to the Discourse Manager for further disposition.

Generally, these speech acts are forwarded to the Prince[8] component, which also has two roles in this architecture: determining what was said and determining what to say. Of course, it does this in conjunction with the Reference component, but Reference has no further say regarding the content of the speech acts once the acts have been handed off to Prince.[9]

Determining What Was Said

Prince determines what was said using a rule-based subsumption architecture. This allows partial parses and those with misrecognized words to still be dealt with efficaciously, without needing an explicit rule that deals with the specific utterance we are processing. For instance, the lowest priority rule matches anything, and returns a speech act that is usually realized as "Huh?" More specific rules might deal with anything that has a reference to an

[8]So in the next version it can be called "the module formerly known as Prince," but also indicative of its status as the homonculus of the system.

[9]Except indirectly: referents whose resolution are dependent on the discourse context may be (directly or indirectly) represented as lambda expressions that return the resolution given a discourse state as parameter. This allows Prince to try different discourse states should the current one be deemed non-fluent without having to go through another pass with Reference.

engine, a city, or a path (that is, specifies an apparent change of state: "from x to y" would be a simple example, though this might also represent a range, of course).

Once the most appropriate rule has been determined (or time to search has run out, and the most appropriate rule found so far is selected), Prince produces a set of strategies to deal with the input. These may range from a set of problem solving strategies (*e.g.*, introduce a new plan involving the mentioned agent), to discourse strategies (*e.g.*, acknowledge the user's greeting). Discourse strategies are dealt with directly by Prince. Problem solving strategies involve a series of requests to the Problem Solving module (see Section 14). Generally, a range of strategies with respect to problem solving is suggested, in priority order. Each is tried in turn, until one is deemed to be adequately coherent by problem solving, and Prince then commits to that strategy.

Determining What To Say

At this point, Prince deals with it's second mission: determining what to say. Given the implementation of the discourse strategy, the results (or problems reported) from the Problem Solver, and the discourse state, Prince generates a series of speech acts that may need to be actualized. These are grouped and marked as being part of a particular problem solver state (that is, the new state of the problem solver if we commit to an update), and a plan. Prince collects these for each speech act in its input incrementally, and then outputs the collection for actualization. While Prince can and should do non-incremental processing, currently the system relies on the illocutionary remapping of the Reference component to make sure adjacent speech acts that should be part of the same "update" are treated as a package.

The Discourse Manager again updates the discourse context based on Prince's output and hands the speech acts on to the Actualization component for final disposition. Actualization is again implemented as a subsumption architecture (for ease of programming this time), partly to mirror our understanding of the speech act implementation in Prince. Rules turn speech acts into display actions, text to be spoken, or both. Each speech act is actualized in turn. Redundant speech acts are (generally) suppressed, as they are an artifact of Prince's current incremental, short memory structure. Actualization keeps track of what we now believe the user to know (*i.e.*, presuming she will recognize the illocutionary force of our utterances) by updating the User Model, which influences further generation and reference.

Finally, Actualization's output is either sent to the SPEECH-OUT module (in the case of generated speech) or applied to the Display Model (in the case of display updates) which translates them into a series of commands for the DISPLAY Module (see Section 11). At this point, a "turn" has been processed (though it may involve several utterances and display actions), and we await further input from the user (that is, another set of speech acts from the Parser).

14 Problem Solver

The Parser described in Section 12 interprets user input and generates logical forms representing the content of what was said (or typed, or clicked, *etc.*). These are interpreted by the Discourse Manager (Section 13) using the context of the conversation to resolve various unspecified or ambiguous elements. As part of this understanding process, the current utterance must be evaluated in the context of the plan or plans under development, and various possible ways of interpreting the utterance examined and compared. This process involves two important research issues being examined in the TRAINS Project, both of which are aspects of intelligent problem solving.

First, as noted above, the interpretation process in a natural mixed-initiative system involves reasoning hypothetically about various alternatives before committing to an interpretation. In addition to this "internal" form of hypothetical reasoning, one of the main goals of mixed-initiative interactive assistants is that they support people doing a similar kind of "external" hypothetical reasoning. That is, they want to examine scenarios, compare them, revise them, undo the changes, refer to previous scenarios, and so on. Both of these forms of hypothetical reasoning require an explicit representation of both the problem solving context and the history of the interactions thus far. One might characterize this functionality as "human-agent" mixed-initiative planning.

The second aspect of problem-solving that we are investigating in the TRAINS Project involves the other end of the problem solving process, which might be termed "agent-agent" mixed-initiative planning. We believe that a robust, realistic planning assistant will need to draw on a variety of knowledge sources and reasoning engines to provide information about aspects of the task. These specialized reasoners can provide fast solutions to hard problems, but must be invoked in an appropriate context, which may differ between reasoners. Their results must then be integrated with respect to the plan and communicated back to the rest of the system.

To address these two issues, we have implemented a Problem Solving module (PS) with the following features:

- Explicit hierarchical representation of plans
- Explicit linear history of the planning process
- Support for hypothetical reasoning with explicit commitment of results
- Integration of separate underlying domain reasoners
- KQML input and output (of course)

The rest of this section describes the main conceptual basis of the current Problem Solver and lists the various requests and responses in detail.

14.1 Problem Solver Principles

In traditional hierarchical planning systems, goals are often simply the undeveloped actions in the hierarchy. Once a plan is fully developed, it is hard to distinguish the actions that

are present because they *are* the goals from the actions that were introduced because they seemed a reasonable solution *to* a goal. In the TRAINS-96 Problem Solving model, the analog of the hierarchical plan is a goal hierarchy. It records the goals and decompositions into subgoals that have been explicitly developed through the interaction with the user. Particular solutions to the goals, on the other hand, are not in the hierarchy explicitly except via the goals that they address. The goals provide an organization of the solution that is useful for many purposes—for revising parts of the plan, for summarizing the plan, and for identifying the focus of the plan for a particular interaction. The goal hierarchy plays a crucial role in managing the focus of attention during the interaction. In general, a user will move up and down the goal hierarchy in fairly predictable ways as they work on one problem and then move to the next.

In general, two distinct aspects must be provided for any plan reasoning operation. The first is an "interpretation" or "hypothetical" mode, where the Problem Solver is asked to evaluate whether an operation makes sense with respect to a proposed focus node. The other aspect is a "update" or "commitment" mode, and involves performing the actual operation at a plan node in order to update the problem solving state. The first capability is not found in most planning systems, but a correlate can be found in the plan recognition literature. In most previous plan recognition systems, however, this distinction was collapsed, and the way that the system checked if a goal was a reasonable interpretation involved building a plan to solve the goal. In our new model, these functions are distinct, and the response to each type of request must address both, returning two separate indicators of success:

- The *recognition score*, which indicates whether the proposed operation is a reasonable thing to request for the focus node; and
- The *answer score*, which indicates whether the system can satisfy the request to perform the operation (*e.g.*, say to find a plan to achieve a new goal).

In the TRAINS-96 system, the recognition score is determined by reasoning about the constraints, and uses information such as constraint consistency to determine the appropriateness of interpretations. The answer score is determined by calling a specialized route planner with the goal statement and checking whether it can find a reasonable solution. There could be other ways of implementing this depending on the domain. The critical issue, however, is that the distinction between these two different capabilities must be supported so that we can identify the right focus node even when we could not perform the requested action.

14.2 Problem Solver Representation

Most Problem Solver requests use action specifications of the form:

(*act-type act-instance-id constraints**)

Constraints within an action are of the form:

(*constraint-name value*)

(:FROM *loc*)	Go from indicated location
(:TO *loc*)	Go to indicated location
(:VIA *loc1 loc2 ... locn*)	Go via indicated locations in order specified
(:AVOID *loc*)	Same as (:NOT (:VIA loc))
(:DIRECTLY *loc1 loc2*)	Go directly from *loc1* to *loc2*
(:DIRECTLY *from-or-to**)	*e.g.*, (:DIRECTLY (:TO CHICAGO))
(:USE *object*)	Ensure that plan uses object in some way
(:NOT *constraint*)	Ensure that the constraint does not hold
(:AND *constraint constraint**)	
(:OR *constraint constraint**)	

Table 17: Possible constraints for action GO

(:LOC *loc*)	Stay at location
(DURATION *N*)	Stay for *N* hours

Table 18: Possible constraints for action STAY

A very simple example, representing going to Chicago, is:

```
(:GO g1223 (:TO CHICAGO))
```

The Problem Solver currently understands only two actions, GO and STAY. The possible constraints on these actions are shown in Tables 17 and 18. The constraints relevant to a particular request are usually specified as the :content argument of the request (not to be confused with the :content of the performative, which specified the entire request).

Several Problem Solver requests also accept parameters specifying two different types of function that can be applied to the operation being requested. Filter functions allow the user to refine solutions based on global properties such as overall time, cost, or in route planning, distance traveled. These functions are specified as:

((*comparison-op value*) *scale*)

where *comparison-op* is one of <, <=, =, =>, >, or =/=, and *scale* is currently one of DISTANCE, DURATION, or COST. For example, "((< 24) DURATION)" is a filter that holds only of solutions that are less than 24 hours in duration.

Preference functions determine the preferred ordering of solutions for a specific goal. They allow the user to prefer plans with minimum or maximum cost, time, and so on. The format of a preference function is:

(*preference-op scale*)

where where *preference-op* is one of `<=` or `=>`, and *scale* is as above. For example, the preference function "`(<= COST)`" would indicate that solutions that minimize cost should be preferred. In addition, regular constraints can be used as filters using the syntax:

`(CONSTRAINT` *constraint*`)`

Thus "`(CONSTRAINT (:VIA Pittsburgh))`" would restrict all solutions to go through Pittsburgh, even if "`(:VIA Pittsburgh)`" was not part of the goal statement. This capability is especially useful for operating on solution sets to pursue alternate solutions without modifying the goals.

14.3 Problem Solver Implementation and Operation

The Problem Solver is implemented as a Common Lisp image. It reads KQML messages from standard input (Lisp stream `*standard-input*`) and writes KQML messages to standard output (Lisp stream `*standard-output*`). The Problem Solver accepts KQML messages of several types:

- Requests to perform interpretation actions
- Requests to perform update (commitment) actions
- Assertions and queries
- Generic messages shown in Table 3 (page 9)

The specific messages in each class will be described separately below. The Problem Solver generates replies to all requests, using an `error` performative to indicate an improper input, `sorry` to indicate inability to satisfy an otherwise reasonable request, and `reply` performatives otherwise.

Interpretation Requests

The set of interpretation requests understood by the Problem Solver (module PS) is shown in Table 19. In general, there are two relationships between goals, which correspond to the refinement and decomposition relations defined in Kautz' model [Kautz, 1987; Kautz, 1991].

- A goal G' is a refinement of a goal G if any plan that satisfies G' also satisfies G.
- A set of goals $G_1, \ldots, G_n$ with temporal constraints, is a decomposition of a goal G, if any plan that satisfies G_1 through G_n and the temporal constraints, also satisfies G.

These relationships are used in the descriptions below.

Performative	Content	Arguments
request	new-subplan	:plan-id :content :filter :preference
request	split-plan	:plan-id :content
request	refine	:plan-id :content :filter :preference
request	do-what-you-can	:plan-id :content :filter :preference
request	extend	:content :plan-id
request	modify	:plan-id :add :delete :filter :preference
request	cancel	:plan-id :content
request	confirm	:plan-id
request	reject-solution	:plan-id
request	select-solution	:plan-id :solution-set
request	confirm	:plan-id

Table 19: Problem Solver (PS) Interpretation Requests

NEW-SUBPLAN: Introduces a new plan as a subplan of the specified plan-id. If it can find a solution to the goal it also returns a solution.

SPLIT-PLAN: Breaks a plan into two subplans with the first having the goal indicated by the action specification. For example, given a plan with goal

```
(GO g11 (:FROM Chicago) (:TO Boston)),
```

the request

```
(SPLIT-PLAN :CONTENT (GO g12 (:TO Pittsburgh)))
```

would break this plan into two subplans: going from Chicago to Pittsburgh (which is not the focused plan), and going from Pittsburgh to Boston.

REFINE: Refine an existing plan with new constraints specified in the action specification. The constraints must be consistent with the old constraints (but not the current instantiation).

DO-WHAT-YOU-CAN: A variant of REFINE that finds the plan in the task tree that best fits the new constraints specified in the action specification. It is useful for getting the problem solver to search for previously introduced plans (say when discourse has lost track of them, or never explicitly knew of them).

EXTEND: Extend an existing plan with new constraints that specify an additional goal. This can be thought of as spawning a "sister" goal node to the current plan. EXTEND on the root node is a special case and it spawns a subgoal.

MODIFY: Modifies an existing plan with new constraints specified in the action specification or constraint list (allowed to handle cases where the action is not known). This differs

from extend in that it may remove old constraints to accommodate the new. For example, consider a route from Chicago to Montreal. An `EXTEND` with constraint `(:TO ALBANY)` would extend the route to Albany from Montreal. A `MODIFY` would remove Montreal as a destination altogether and construct a route from Chicago to Albany.

`CANCEL`: Tries to modify the plan by removing the action/goal/object specified in the content. The content is of the form `(GOAL` *action*`)` or `(OBJECT` *object*`)`. For example, "Cancel the route from Avon to Bath" has content

```
(:OBJECT (:ROUTE r1 (:FROM Avon) (:TO Bath)),
```

while "Cancel the goal of getting to Corning" would be

```
(:GOAL (:GO g1 (:TO Corning)).
```

Actions are treated as objects, so "Cancel going from Avon to Bath" would have the content

```
(:OBJECT (:GO g1 (:FROM Avon) (:TO Bath)).
```

When cancelling goals, the `PLAN-ID` gives a preference, but if the goal description doesn't match the content, this will look for other plans that do. It returns a result of form:

```
(DELETE-GOAL :PLAN (PLAN4597 :GOAL NIL :AGENT NIL :ACTIONS NIL))
```

(see below concerning replies).

`CONFIRM`: Interprets a confirmation by the user with respect to the problem solving state. Returns an update code that can be used to accept this interpretation.

`REJECT-SOLUTION`: Interprets a rejection act by the user and attempts to find an alternate solution. Returns an update code that can be used to accept this interpretation.

`SELECT-SOLUTION`: Installs a new solution set (and presumably a new solution) with the indicated plan. This will succeed only if the solution actually satisfies the objective of the plan.

All interpretation requests generate a `reply` whose content is as follows:

```
(answer :recognition-score S_r :answer-score S_a :ps-state ID
        :result (flag display-info)
        :reason (reason :type type :info info :msg msg))
```

Performative	Content	Arguments
request	update-pss	:plan-id :ps-state
request	clear	
request	new-scenario	:content
request	new-problem	:content
request	delete-plan	:plan-id
request	undo	:ps-state

Table 20: Problem Solver (PS) Update Requests

As described above, the recognition score, S_r, is an indication of whether the speech act was interpretable in the given context or not. Possible values are `:good`, `:ok`, `:bad`, or `:impossible`. The answer score, S_a, uses the same values to indicate whether or not the request could be satisfied. The `:ps-state` *ID* can be used to accept (commit to) this interpretation later. The values of the `:result` argument depend on the particular request, and the *display-info* contains information of use in generation (such as cities, engines, *etc.*). The `:reason` argument is used to explain all scores except `:good`. For more details, see [Allen, 1996b].

Update Requests

The set of update requests understood by the Problem Solver (module PS) is shown in Table 20. Update requests actually change the Problem Solver's hierarchical plan representation, typically using a `ps-state` identifier that was returned by an interpretation action. These requests also generate a `reply` whose `:content` is an answer list. The format of the `:result` in the answer depends on the act. Generally, the answer also specifies the name of the current `ps-state` and the plan node (`plan-id`) affected by the update.

`UPDATE-PSS`: Update the current problem solving state with the update returned in the `ps-state` slot from a previous query. The result is "`(PSS-UPDATE :SUCCESS)`" if successful. The `plan-id` slot in the answer specifies the node modified or created by the update. Since update should always succeed, any other response will be an `error` or `sorry` performative.

`CLEAR`: Reset the problem solving state, leaving the initial scenario unchanged. Typical realization of "Let's start over." Result is of form:

`(ANSWER :RESULT (:RESET :PLAN` *root* `:PS-STATE` *state*`))`

`NEW-SCENARIO`: Reset the problem solving state and the scenario (*i.e.*, the map information). The scenario specification is an assertion as defined below. The result is the same as for `CLEAR`.

`NEW-PROBLEM`: Keeps the current map information, but defines new engines and problems. Result is the same as for `CLEAR`.

Performative	Content	Arguments
assert	*prop*	
ask-if	*prop*	:plan-id :ps-state
ask-one	*prop*	:plan-id :ps-state :aspect
ask-all	*prop*	:plan-id :ps-state :aspect
ask-about	*expr*	:plan-id :ps-state

Table 21: Problem Solver (PS) Knowledge Base Requests

DELETE-PLAN: Deletes the indicated plan from the problem solving tree.

UNDO: Backs up the problem solving state to the point where the given `ps-state` was returned by `UPDATE-PSS`. If `ps-state` is not specified, this pops the Problem Solver history one step.

Assertions and Queries

The Problem Solver manages queries about the problem solving state, and passes on queries to the appropriate domain reasoners (currently integrated with the Problem Solver as one module). The set of knowledge base messages understood by the Problem Solver is shown in Table 21.

To assert information, you simply use a KQML `assert` performative with the `:content` set to the information to be asserted. No variables are allowed in assertions. For example, to assert where an engine is we would send:

```
(assert :content (:and (:type :engl :engine) (:at-loc :engl :chicago)))
```

The response would be:

```
(reply :content (answer :result :success))
```

An assert currently only fails if it is ill-formed, in which case an `error` performative is sent in reply.

There are several different forms of queries. All queries may specify a `:plan-id` and a `:ps-state` field, defining the plan context and temporal context of the query. If these are not specified, default values are used making the query relative to the last plan updated and the current time. Queries that involve the entire problem solving state or that are handled by the domain reasoner(s) will ignore these values when they are not applicable. Variables in queries are simply atoms starting with a question mark, *e.g.*, "`?ENG`" is a variable that will match any expression.

All queries result in a `reply` with `:content` of the form

(`answer :result` *prop* `:vars` (*v1 v2* ...))

where the `:result` is particular to each type of query and `:vars` supplies information about the variables used in the `:result`. If the query cannot be interpreted, then an `error` message is generated.

`ASK-IF`: This performs a yes-no query on the database, which returns a `:result` of `T` if true, `NIL` if false, and `:unknown` if it is undetermined. That is, a successful query would return:

```
(reply :content (answer :result t))
```

`ASK-ONE`: This looks for an expression that makes the *prop* true. It returns a result consisting of instantiations of the variable list for the variables of interest. For example, the query

```
(ask-one :content (:problems ?type ?loc ?delay) :aspect (?loc ?delay))
```

asks for the location and delay of a problem and a plausible result would be

```
(reply :content (answer :result (chicago 5) :vars (?loc ?delay)))
```

That is, `?loc` is `chicago` and `?delay` is 5. If the `:aspect` in the query is `ALL`, then values for all variables in the query are returned. For example,

```
(ask-one :content (:problems ?type ?loc ?delay) :aspect all)
```

might return

```
(reply :content (answer :result (:wind chicago 5)
                        :vars (?type ?loc ?delay)))
```

`ASK-ALL`: This looks for all expressions that make the *prop* true. It returns a result that is a list of all possible bindings for the variables of interest. For example, the query

```
(ask-all :content (:problems ?type ?loc ?delay) :aspect (?loc ?delay))
```

asks for the location and delay of all problems. The result might be

```
(reply :content (answer :result ((chicago 5) (buffalo 4))))
```

If the `:aspect` is set to `ALL`, then values for all variables in the query are returned (see example for `ASK-ONE`).

`ASK-ABOUT`: This will generate a `reply` whose content is a list of all propositions that contain the *expr* as an argument.

Further details regarding the Problem Solver knowledge base functions and the set of predicates understood by the current version of the module can be found in [Allen, 1996b].

15 Other Modules

This section describes a variety of other modules making up the TRAINS-96 System. Most of them provide services involved in making TRAINS a useful experimental and demonstration system. Although they do not themselves address the research issues being explored in TRAINS, they make such exploration possible and their presentation here illustrates both the range of such services that are needed and the flexibility of the TRAINS System, particularly the communication infrastructure, in supporting them.

15.1 Speech Controller (SPEECH-X)

The TRAINS-96 Speech Controller, module SPEECH-X, is responsible for starting and stopping speech recognition and displaying its results. This allows the SPEECH-IN (Section 7) and SPEECH-PP (Section 8) modules to be implemented as "faceless" modules that do not have to worry about X-Windows displays and the like. A sample display from the Speech Controller is shown in Figure 8. The complete set of messages understood by it is shown in Table 22.

When the "Click to Talk" button is pressed, the Speech Controller sends a `start` request to SPEECH-IN to initiate recognition. It then monitors the output of both SPEECH-IN and SPEECH-PP (using `listen` requests to the Input Manager) and displays their results (`tell word` and `tell backto`) in the top and bottom text windows, respectively. When the Speech Controller button is released, it sends a `stop` message to SPEECH-IN to stop further recognition. The Speech Controller can be operated in either click-and-hold mode, as described above, or in a click-to-talk mode where the first click starts recognition and the second one stops it. See the manpage (Section D.23, page 133) for details.

15.2 Startup Screen (SPLASH)

The TRAINS-96 Startup Screen module may seem somewhat silly at first glance. Its main purpose is to display a fancy picture of a train and allow the user to start a conversation by clicking the "Start" button. In fact, although it is quite simple, it serves two important purposes: synchronizing all the other modules at the start of a conversation, and recording user name, language, sex, and options.

The Startup Screen module understands the basic messages listed in Table 3 (page 9). When the "Start" button is pressed, it broadcasts a `start-conversation` message containing the user information. Note that this is one of the few true KQML broadcasts (*i.e.*, a `broadcast` performative) used in the system. It then hides its window until an `end-conversation` is received, at which point it redisplays itself, allows the user to save the session or send email to the maintainers. It then waits for the user to start the next session by pressing the "Start" button again. The "Quit" button sends an `exit` request to the Input Manager, which shuts down the entire system.

The Startup Screen module is also responsible for creating the log directory and sending `chdir` messages to alert other modules. It also writes the file "`user`" in the log directory, containing the user information.

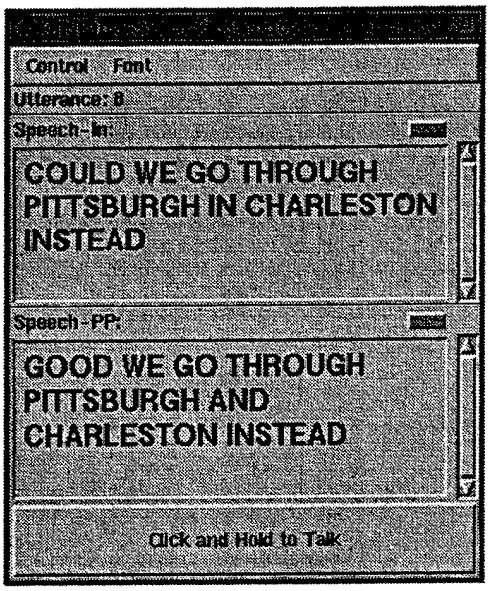

Figure 8: TRAINS-96 Speech Controller Display

Performative	Content	Arguments
`tell`	`start`	`:uttnum`
`tell`	`input-end`	`:uttnum`
`tell`	`end`	`:uttnum`
`tell`	`word`	*word* `:uttnum :index`
`tell`	`backto`	`:uttnum :index`
`request`	`set-button`	
`request`	`unset-button`	
`request`	`exit`	*status*
`request`*	`chdir`	*dirstring*
`request`	`hide-window`	
`request`	`show-window`	
`tell`	`start-conversation`	`:name :lang :sex`
`tell`	`end-conversation`	

Table 22: Speech Controller (SPEECH-X) Messages (* indicates message is ignored)

Performative	Content	Arguments
tell	start	:uttnum
tell	input-end	:uttnum
tell	end	:uttnum
tell	word	*word* :uttnum :index
tell	backto	:uttnum :index
tell	mouse	*objects*
tell	confirm	*tag*
request	log	*text*
request	exit	*status*
request	chdir	*dirstring*
request	hide-window	
request	show-window	
tell	start-conversation	:name :lang :sex
tell	end-conversation	

Table 23: Transcript (TRANSCRIPT) Messages

15.3 Transcript (TRANSCRIPT)

The TRAINS-96 Transcript module's role is to gather a somewhat human-readable transcript of a TRAINS session, such as might be useful for including in a paper or presentation. It also provides an X-Window display of the transcript, which is especially useful during debugging and replay.

The Transcript module understands the messages listed in Table 23. These are essentially the messages broadcast (selectively) by the speech recognition modules, as well as some messages broadcast by the Display in response to user mouse actions, and the common messages from Table 3 (page 9).

The Transcript module monitors the speech recognition and Display modules (using Input Manager `listen` requests) and transcribes the user's input from these messages. The `log` request is provided for modules to add lines to the transcript directly. For example, the Discourse Manager uses it to add indications about generated speech and map displays in a more readable form.

The Transcript module writes the file "`transcript`" in the log directory, and opens a new transcript file upon receipt of a `chdir` request.

15.4 Scenario Chooser (SCENARIO)

The TRAINS-96 Scenario Chooser is a simple module which presents a panel from which the user can select various scenarios. This makes it simple to repeat a scenario, run a preset (although not "scripted" scenario), or change the complexity of the random scenarios.

In addition to the common messages shown in Table 3 (page 9), the Scenario Chooser also accepts a request of the form:

```
(request :content (define :label string :content expr))
```

This adds a new preset scenario whose label (button) is *string* and which, when selected, causes the given *expr* to be sent as the `:content` of a request to the Discourse Manager. Of course, this is highly-dependent on the fact that it is the DM that understands how to set a scenario, but ...

In operation, clicking on an item in the Scenario Chooser causes the appropriate message to be sent to the Discourse Manager. It is recommended that this only be done between sessions (*i.e.*, when the splash screen is displayed). As well, if a preset scenario is selected, then upon receipt of an `end-conversation` message, the Scenario Chooser will automatically move to the next scenario and send the appropriate message. This makes it easy to preset a sequence of scenarios for an extended session.

The Scenario Chooser reads its initial set of presets from a file named `scenario.rc` in the `etc` directory of the TRAINS tree. See the manpage (Section D.18, page 119) for more details.

15.5 Shortcuts Panel (SHORTCUT)

The TRAINS-96 Shortcuts Panel is similar to the Scenario Chooser described in the previous section. Rather then sending specific messages based on random and preset scenarios however, the Shortcuts Panel allows arbitrary messages to be sent by clicking on them.

In addition to the common messages shown in Table 3 (page 9), the Shortcuts Panel also accepts a request of the form:

```
(request :content (define :label string :content perf))
```

This adds a new item whose label (button) is *string* and which, when selected, causes the given performative (*perf*) to be sent. One useful message is to send the performative

```
(request :receiver IM :content (exit 0))
```

and so shutdown the system. Another useful one is:

```
(broadcast :content (end-conversation))
```

to abort a session. The Shortcuts Panel allows new shortcuts to be created and edited, and allows the shortcuts to be saved to a file.

The Shortcuts Panel reads its initial set of shortcuts from a file named `shortcut.rc` in the `etc` directory of the TRAINS tree. See the manpage (Section D.20, page 123) for more details.

15.6 Sound Effects (SFX)

The TRAINS-96 Sound Effects module is actually even simpler than its name implies. In response to a message of the form

```
(request :content (play "filename"))
```

it simply opens the named file and sends its contents to the AudioFile server for playback. The file is assumed to contain data suitable for the AudioFile server (currently 16-bit, 16000 Hz, linear PCM-encoded data, such as recorded by the Speech recognition module). When the audio has finished playing, a reply with content `done` is generated. The Sound Effects module also understands the common messages shown in Table 3 (page 9) although it ignores all of them aside from `exit`.

Originally this module was conceived as something of a joke, nominally to allow "mood music" (or alert noises) to be played by the system during a conversation. It turned out to be useful during replay to play the user's utterances, and could be more useful as we explore multi-modal interaction in more detail. It could also be easily enhanced, for example to understand different types of audio files, adjust volumes, *etc.*

15.7 Parse Tree Viewer (PVIEW)

The TRAINS-96 Parse Tree Viewer, module PVIEW, provides a simple X/Motif interface with which the user can examine parse-trees during a dialogue. This viewer can help developers working on the parser as well as helping a user understand how the system understood their utterances. The Parse Tree Viewer module understand the common messages from Table 3 (page 9).

When the user presses the button on the Viewer's panel, a message of the form

```
(request :content (parse-tree) :receiver parser)
```

is sent to the TRAINS-96 Parser requesting the most recent parse tree. The Parser replies with a `parse-tree` message containing a summary of the best syntactic analysis covering the most recent turn in its entirety. If several hypotheses are equally rated, then the Parser provides a set of equally qualified parse trees. The Parser uses its own notion of what makes a *best* parse. If the turn consists of more than one utterance, one analysis/tree per turn will be provided. The analysis is displayed in the Parse Tree Viewer window and is appended to the display each time the user presses the button. The display is not automatically updated after each turn, since we assume that user will decide when tree viewing is needed.

The expression returned in the `parse-tree` message consists of a parenthesized list of non-terminal and terminal entries. Non-terminal entries have the form:

```
"NTX" NT 1 NTI 2 NTJ 3 NTK
```

where:

- "NTX" is a quoted node-name consisting of a concatenated node-type and number,
- NT is the node-type for the preceding node-name,
- NTI is the first daughter (also a node-name) of the preceding node-name,
- and the successors are the second, third, etc. daughters of the preceding node-name.

These daughter node-names are defined by subsequent entries in the parse tree. Terminal entries have the form:

```
"NTX" NT LEX W
```

where "NTX" is a quoted node-name and NT is the node-type for the node-name (as for the non-terminal entry), and W is the word for the terminal node.

Here is an example (the spacing and indentation shown are provided only to clarify and are certainly not mandatory):

```
(tell :content (parse-tree
  ("UTT6869" UTT 1 PUNC6820 2 UTT6862 3 PUNC6864
     "PUNC6820" PUNC LEX START-OF-UTTERANCE
     "UTT6862" UTT 1 S6861
       "S6861" S 1 VP6860
         "VP6860" VP 1 V6832 2 PATH6858
           "V6832" V LEX GO
           "PATH6858" PATH 1 ADVBLS6857
             "ADVBLS6857" ADVBLS 1 ADVBL6855
               "ADVBL6855" ADVBL 1 ADV6841 2 NP6852
                 "ADV6841" ADV LEX TO
                 "NP6852" NP 1 NAME6849
                   "NAME6849" NAME LEX BOSTON
     "PUNC6864" PUNC LEX END-OF-UTTERANCE) ))
```

After receiving such a message from the Parser, the Viewer displays a tree like that shown in Figure 9. The terminals (leaves of the tree) are shown in blue on the display.

15.8 Replay

The TRAINS-96 Replay Facility is one of the most useful aspects of the TRAINS-96 system. It is not really a module itself—instead, it is a version of the system running in replay mode, together with a REPLAY module that drives the other modules through a recorded session. It provides real-time playback, as well as "tapedeck-style" buttons for stepping back and forth through the dialogue.

The first part of the Replay Facility is a version of the system startup script that launches only those modules needed for replay and provides the necessary arguments for replay mode. For example, the Input Manager is started with the -nolog option to prevent it writing a

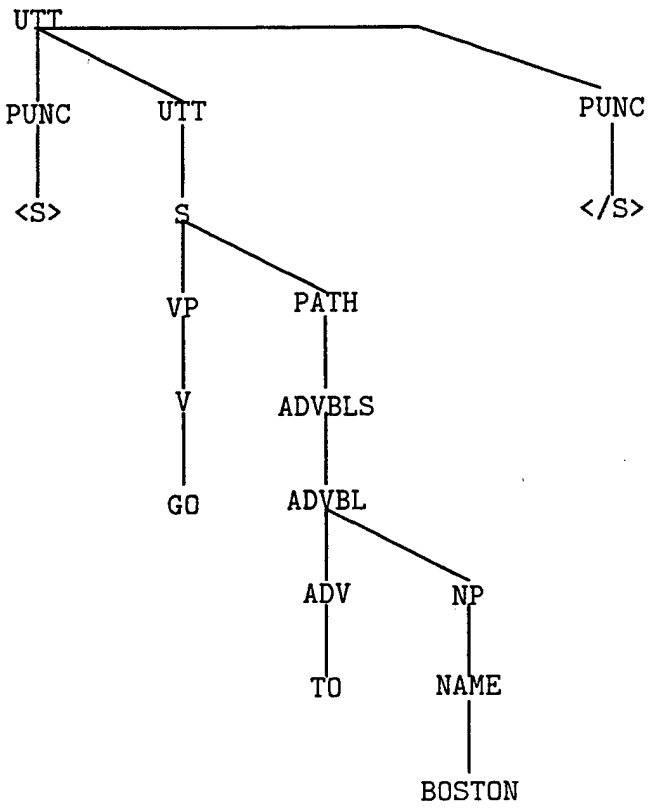

Figure 9: Sample Parse Tree Viewer Display

log (which would simply repeat the recorded log). And many modules, such as the Parser, Discourse Manager, Speech Recognizer, and Speech Post-Processor are not needed during replay. Rather, their functions are recreated from the recorded session.

The second part of the Replay Facility is a module that reads a previous Input Manager log, extracts the messages, and replays them (most of them, anyway) in real-time. For example, DISPLAY commands cause the display to be updated as in the original session, and requests for Speech Generation are resent to the SPEECH-OUT module for regeneration. The replay module also arranges to play the user's utterances (recorded by SPEECH-IN) at the appropriate time using the Sound Effects module. The result is a recreation of the screen displays (including the Speech Controller with its incremental display of speech recognition) and audio for an entire session.

15.9 Input Manager Utilities

A collection of Input Manager utilities have proven to be very useful during the development and operation of TRAINS-96. Since the TRAINS-96 Input Manager uses socket-based communication, it can be tricky to connect a process to the Input Manager, and hence into the running system. As described in Section 5, the Process Manager is designed to make this simpler, by looking after connecting any processes that it launches. However, especially during debugging, it can be helpful to be able to inject messages into the system without starting an entire Process Manager process.

The following four utilities provide this functionality. They are further described in their manpages (Sections D.7–D.10, pages 96–99).

`tim_msg`: Formats its arguments as a performative, then connects to the Input Manager and sends it the performative. Exits, closing the IM connection, after sending the message.

`tim_cat`: Connects to the Input Manager and then copies its standard input to the IM. Exits, closing the IM connection, after end-of-file on standard input.

`tim_client`: Connects to the Input Manager and then sends input from its standard input to the IM and messages from the IM to stdout. If a `register` message is sent, this program can send and receive messages just like any other module in the system. Exits, closing the IM connection, after end-of-file on standard input.

`tim_exec`: Connects to the Input Manager and then calls `exec(2)` with its arguments to launch a new process with its standard input and standard output connected to the IM. This is most of what the Process Manager does, as it turns out.

All four utilities connect to the Input Manager on the local host at the "well-known" port, unless the environment variable `TRAINS_SOCKET` is set to specify a different "host:port". See Section 4 regarding the Input Manager for more details.

15.10 Dialog Archiving Tools

We have mentioned how several of the TRAINS system modules log their activity in per-session log files. These log files are saved in a directory created by the Splash Screen module for each dialogue, as described in Section 15.2. Log directories are created in the directory given by the `TRAINS_LOGS` environment variable, or in `$TRAINS_BASE/logs` by default. Unique names for the log directories are constructed using the current date and time.

This section briefly introduces available tools for organizing and summarizing the contents of log directories. A few more details are available in their manpages (Sections D.1–D.3, pages 77–80).

`dlg_org`: This is a Perl script that organizes the contents of the log directory given on the command-line. It compresses large log files, such as the DM debugging logs, and it categorizes the logs and data into several sub-directories:

- **`data`:** All audio (`*.au`) and search results (`*.out`) files from Sphinx-II are placed here.
- **`mfc`:** This sub-directory is initially empty, but MFC parameter files will be placed here by subsequent speech experiments.
- **`s2`:** This sub-directory is initially empty also, but Sphinx-II log files will be placed here by subsequent speech experiments.
- **`sent`:** This sub-directory is initially empty, but hand transcriptions of the dialogue data will be placed here.
- **`sys`:** All system module logs and the transcript are placed here.

`dlg_check`: Checks and summarizes the contents of the dialogue directory (given on the command-line) that was recorded during a session with the TRAINS-96 system. Its summary is based primarily on the contents of the session transcript, and provides the user information and the number of utterances of various types. If necessary, `dlg_check` first runs `dlg_org` to organize the contents of the dialogue into suitable sub-directories.

`dlg_play`: Plays the utterances in the given dialogue directory one at a time. The user is prompted before each utterance with several options: `repeat`, `back-up`, `quit`, or `continue`. It is also possible to type `n` followed by a number to jump to the utterance in that position. This should be thought of as a poor-man's Replay.

References

[Allen, 1995] James F. Allen, "The TRAINS-95 Parsing System: A User's Manual," TRAINS technical Note 95-1, Department of Computer Science, University of Rochester, Rochester, NY, 14627, September 1995.

[Allen, 1996a] James F. Allen, "Logical Form in the TRAINS-96 System," Trains technical note, Department of Computer Science, University of Rochester, Rochester, NY, 14627, 1996. To appear.

[Allen, 1996b] James F. Allen, "Problem Solving Manager Documentation". TRAINS Project Online Documentation, 1996.

[Allen and Schubert, 1991] James F. Allen and Lenhart K. Schubert, "The TRAINS Project," TRAINS Technical Note 91-1, Department of Computer Science, University of Rochester, Rochester, NY, 14627, May 1991.

[Allen *et al.*, 1995] James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung Hee Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio, and David R. Traum, "The TRAINS Project: A case study in defining a conversational planning agent," *Journal of Experimental and Theoretical AI*, 7:7–48, 1995. Also available as TRAINS Technical Note 93-4, Department of Computer Science, University of Rochester.

[Ferguson *et al.*, 1996] George Ferguson, James Allen, and Brad Miller, "TRAINS-95: Towards a Mixed-Initiative Planning Assistant," in *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pages 70–77, Edinburgh, Scotland, 29–31 May 1996.

[Finin *et al.*, 1993] Tim Finin, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzson, Donald McKay, James McGuire, Richard Pelavin, Stuart Shapiro, and Chris Beck, "Specification of the KQML Agent-Communication Language". Draft, 15 June 1993.

[Heeman and Allen, 1995] Peter A. Heeman and James F. Allen, "The TRAINS-93 Dialogues," TRAINS Technical Note 94-2, Dept. of Computer Science, University of Rochester, Rochester, NY, March 1995.

[Hinkelman and Allen, 1989] Elizabeth A. Hinkelman and James F. Allen, "Two constraints on speech act ambiguity," in *Proceedings of the Twenty-Seventh Annual Meeting of the Association for Computational Linguistics (ACL-89)*, pages 212–219, Vancouver, BC, 25–27 June 1989.

[Huang *et al.*, 1992] Xuedong Huang, Fileno Alleva, Hsiao-Wuen Hon, Mei-Yu Hwang, and Ronald Rosenfeld, "The SPHINX-II Speech Recognition System: An Overview," Technical Report CS-92-112, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, January 1992.

[Kautz, 1987] Henry A. Kautz, *A Formal Theory of Plan Recognition*, PhD thesis, Department of Computer Science, University of Rochester, Rochester, NY, May 1987. Available as Technical Report 215.

[Kautz, 1991] Henry A. Kautz, "A Formal Theory of Plan Recognition and its Implementation," in *Reasoning about Plans*, pages 69–126. Morgan Kaufmann, San Mateo, CA, 1991.

[Levergood *et al.*, 1993a] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart, "AudioFile: A network-transparent system for distributed audio applications," Technical Report 93/8, Digital Equipment Corporation, Cambridge Research Lab, Cambridge, MA, 11 June 1993.

[Levergood *et al.*, 1993b] Thomas M. Levergood, Andrew C. Payne, James Gettys, G. Winfield Treese, and Lawrence C. Stewart, "AudioFile: A network-transparent system for distributed audio applications," in *Proceedings of the USENIX Summer Conference*, June 1993.

[Ringger, 1995] Eric K. Ringger, "A Robust Loose Coupling for Speech Recognition and Natural Language Understanding," Technical Report 592, Department of Computer Science, University of Rochester, Rochester, NY, 14627, September 1995.

[Ringger and Allen, 1996] Eric K. Ringger and James F. Allen, "A fertility channel model for post-correction of continuous speech recognition," in *Proceedings of the Fourth International Conference on Spoken Language Processing (ICSLP'96)*, Philadelphia, PA, October 1996.

[Searle, 1969] John R. Searle, *Speech Acts: An essay in the philosophy of language*, Cambridge University Press, Cambridge, England, 1969.

[Sikorski and Allen, 1996] Teresa Sikorski and James F. Allen, "TRAINS-95 System Evaluation," TRAINS Technical Note 96-3, Dept. of Computer Science, University of Rochester, Rochester, NY, July 1996.

A Running the TRAINS System

This section is intended as a very brief introduction to running the TRAINS-96 System on our facilities at Rochester. Many other permutations and combinations are possible—see the manpages that follow and the module descriptions themselves for further details.

The steps required to run the system are, basically:

- Setup the environment
- Start the Input Manager and Process Manager
- Start other modules by sending messages to the PM via the IM

These steps are considerably simplified by the `trains` script whose manpage is in Section D.16 (page 115). You may want to refer to a copy of that script in order to understand what follows.

All pathnames in the TRAINS system are interpreted relative to an environment variable `TRAINS_BASE`. For the current (as of this writing) version of the TRAINS-96 system, the default base of the TRAINS directory tree is:

```
/u/trains/96/2.0
```

That's where you want to look for executables (sub-directory `bin`) and manpages (sub-directory `man`), among other things.

A.1 Setup Environment

The `trains` script uses a set of environment variables to customize the execution of the system. These are detailed in the manpage so I won't repeat them here, but they include such things as finding a useful value for the `DISPLAY` variable if the user hasn't set one, arranging for audio input and output on the appropriate machine, and so on.

As well, the `trains` script starts any servers required by the system but which function outside the control of the system. In particular, it starts the AudioFile audio server, `Asparc` or `Asparc10`, and arranges to kill it when it exits. This would not be necessary if everyone ran an audio server the way they do an X server, but they don't so we do, if you follow me ...

A.2 Start IM and PM

The next thing to do is get the components of the TRAINS system running, but here we are faced with a chicken-and-egg problem. So the `trains` script first launches the Input Manager (`tim`), and arranges to kill it when the script exits. After a short delay to allow the IM to initialize its socket connection, the `trains` script launches the Process Manager (`tpm`), and also arranges to clean it up. The PM will establish a connection to the IM, whereafter it is ready to receive messages.

A.3 Start Other Modules

To start the remaining modules, the `trains` script uses the `tim_cat` program to "inject" messages into the system. As described in its manpage (Section D.7, page 96), `tim_cat` connects to the Input Manager and then copies its standard input to the IM. The `trains` script provides a set of `start` messages for the Process Manager (*i.e.*, with `:receiver PM`) as standard input for the `tim_cat` process. As these are received, the specified modules are launched and connected into the system.

With this setup, one can change the set of modules launched by the system by editing the `trains` script and modifying the set of messages sent using `tim_cat`. This can be useful if only a subset of the modules are needed, or if the parameters for a particular module need to be changed. For example, a different executable can be specified for a module by changed the `:exec` parameter in the `start` request that launches it, or a module's command-line arguments can be specified using the `:argv` parameter.

It would be nice to have a more friendly way to do this, but for now that's how it works. At least it's better than TRAINS-95, where the modules and their parameters were hard-coded into the Process Manager!

B Speech Lab Setup

This section describes the care and feeding of the speech lab equipment for use with TRAINS-96.

B.1 Introduction

The current version of the TRAINS system uses a single SPARCStation for both audio input and output, together with the speech lab audio rack. The main issues are:

1. Amplifying microphone input to line levels for speech recognition.
2. Amplifying and mixing SPARC audio output for speech generation.
3. Providing audio feedback in the user headset so they can hear both themselves and the system.
4. Keeping system output out of the speech recognition input, to prevent the system trying to understand itself.

The following setup addreses these issues.

B.2 Audio Rack Setup

Figure 10 shows the front and back panel configurations for the speech lab audio rack equipment. I have labelled most of the cables with colored tags (indicated in the figure).

Starting from the headset, it is plugged into the Whirlwind "black box", which splits the special six-pin headset cable into a balanced mic output and a phono plug headphone input.

The mic output goes to the preamp **In2** (either channel would do) via the *green* cable. The preamp output for that channel (**Out2**) goes via an impedance splitter/matcher into the mixer **CH2-R** (any channel would do).

The other audio input to the mixer is from the headphone jack of the SPARCStation being used for audio. It goes to the mixer **CH1-L** via the *yellow* cable (any other mixer channel would do, so long as it uses the other side of the stereo split from the microphone input).

There are three outputs from the mixer:

1. The mixer **Mono** output goes to the black box headphone input, to provide the user with both their own and the system's audio. I used the *purple* cable for this.
2. The mixer **Stereo-R** output goes to the **LineIn** input of the SPARCStation being used for audio via the *red* cable.

If you swapped the input channels from the mic and SPARC jack, of course you'd need to swap the outputs accordingly. The SPARC LineIn should only get microphone audio.

Finally, the external speakers are driven directly from the SPARC LineOut jack, using their permanently-attached cable. The right speaker plugs into the back of the left one.

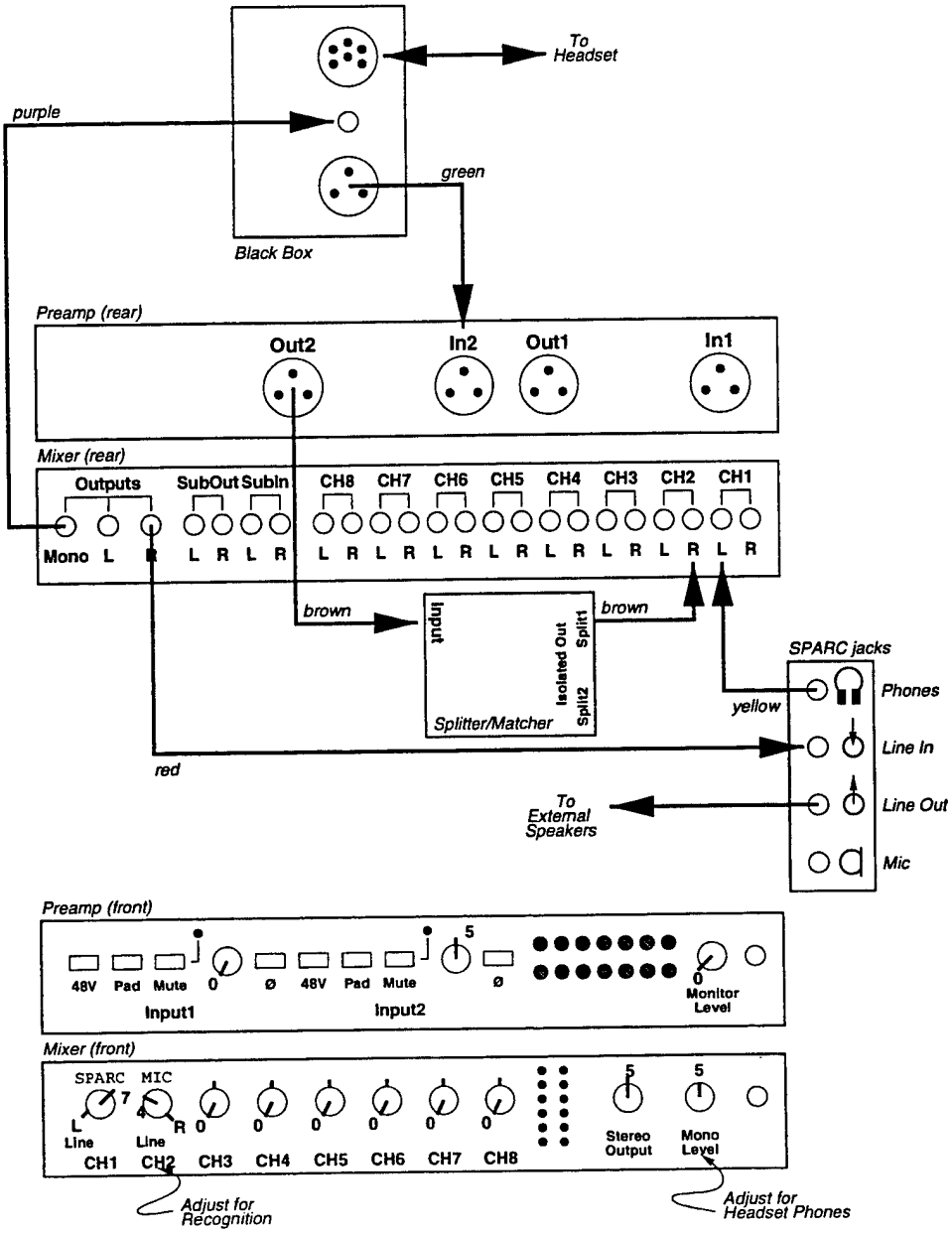

Figure 10: Speech lab audio rack front and back panels

B.3 Audio Rack Settings

With the rack cabled as shown in Figure 10, you now need to adjust the front panel settings on the rack. The figure shows settings that I have found to work well, the details are in what follows.

The preamp is easy. You need a decent amount of gain on the mic, but not so much that the noise from the computer equipment is overwhelming. I have found **5** to be a decent compromise (it makes the leds blink entertainingly without drowning you in hiss).

The mixer settings are a bit trickier, since they interact with the SPARC audio device settings. These descriptions assume you used the cabling setup described in the previous section.

The SPARC audio channel (**CH1**) should be panned completely to the **Left**. I have found that with the audio output level set at 70% (via the Audio manager), this level needs to be about **7**. Watch the leds on the mixer and see.

The mic input channel (**CH2**) should be panned completely to the **Right**, and the level adjusted to give decent flashing of the leds on the mixer. I have found that **4** seems about right. You can trade-off between the preamp gain and this level setting if you like. You want decent levels but not too much hiss. Watch the mixer leds.

The mixer outputs are the key to the whole thing. The **Mono Level** feeds the headset both the SPARC and mic audio. The level should be adjusted to make the volume in the headphones comfortable. The relative mix of SPARC and mic is determined by the two input levels. They should be roughly equal as shown on the mixer output level meter when both the human and the system are talking.

The **Stereo Output** feeds the SPARC audio input for speech recognition. **The most important thing is to get a reasonable level for speech recognition!** I have found that this means quite a low output level, like around **4 or 5**, when the SPARC record level is set to 70 (using the Audio manager). Presumably much more will cause the SPARC A/D converter to clip. If you had a soft talker, you might need to turn this up. You could also adjust the SPARC record level using the Audio Manager, or up the preamp or mixer gains for the microphone input.

The volume at the external speakers should be adjusted using the separate volume control on the speakers. In extreme cases the SPARC audio output level can be adjusted using the Audio manager, but note that this affect the balance in the headset unless the mixer **CH1** is also adjusted.

C Travel System Instructions

This section describes the care and feeding of the TRAINS travel system. It may be somehwat obsolete in that it describes a two-machine travel system while the latest version of TRAINS-96 can run on a single machine. With a suitable reinterpretation, most of the comments still apply, although you will need to adjust the `trains` script for the different setups.

C.1 System Overview

The current incarnation of the TRAINS travel system involves a Sparc10 workstation (`larynx`) with external 9G drive, an UltraSparc workstation (`micro`) with external 4G drive, a 17-inch monitor, one Sparc audio box, two keyboards, a Sony DATman, and assorted microphones, cables and cords.

The TRAINS system can run on any combination of the two workstations, both of which boot from their external drives to run standalone. The workstations are connected using a twisted-pair direct-connection (reversed) 10Base-T cable. The DATman is used as a preamp for the microphone headset.

This document describes a configuration in which the monitor is connected to `larynx` but the audio inputs are connected to `micro`. This has been found to provide the best quality audio input and output. Should you want to run with audio on `larynx` (the original standard configuration), simply make connections to the jacks on `larynx`'s speakerbox rather than `micro`'s back panel (and adjust the `trains` script).

C.2 Before You Go

Here are some things to think about before packing and shipping the system:

1. You need a bootable disk for the standalone system. The staff can help you with this if it isn't done already.

2. Obviously, the external drive needs to have all the relevant code for the system in the appropiate place. The Sparc10 can be connected to the department network and the external drive mounted separately, allowing files to be transferred back and forth easily. To mount the /u parition temporarily, use the command

   ```
   % sux mount /dev/sd2g /mnt
   ```

   You need superuser privileges to do this, naturally.

3. You need return airbills to put on the cases for shipping the equipment back. Pat needs to fill these out in advance. Make sure to check that appropriate shipping options have been specified (*e.g.*, next-day or slower shipping).

4. Keyboards should be wrapped in a plastic bag or something, to prevent their keycaps from falling off and getting lodged inside the cases.

5. Unless something changes, the cases cannot be locked.

C.3 Workstation Setup

This section describes how to get the two workstations and the external drive hooked up and running. The next section describes how to hook up the audio components.

Figure 11 shows the back panel configuration for the various machines. The following is a blow-by-blow description of how everything goes together. Don't turn anything on until you've read all the instructions.

1. Make sure the powerbar is unplugged and turned off so that nothing starts until you're ready.

2. Hook up `larynx`'s power cord, keyboard, optical mouse, and speaker box. Note that the speaker box connects to a small parallel port using the strange looking cable with an adapter in the middle of it. For some reason, Sun decided to include an ethernet input on that connector, so the icon next to the connector is funny and the adapter splits the ethernet connection off. The mouse plugs into the keyboard, which itself plugs into `larynx` with the obvious cable.

3. Hook the bottom SCSI port on the 9GB drive to `larynx`'s bottom SCSI port using the short 50/25 SCSI cable. Leave `larynx`'s top SCSI port (with the status LED) unconnected. **Terminate** the other (top) SCSI port on the 9GB drive with the large (50-pin) terminator. Hook up the 9GB drive's power cable.

4. Hook up `micro`'s power cord and keyboard. The keyboard's cable is permanently attached to it, and the mechanical mouse is optional but not necessary.

5. Connect `micro`'s SCSI port to the top port of the 4GB drive using the long 25/25 SCSI cable. **Terminate** the bottom port on the 4GB drive using the small (25-pin) terminator. Connect the 4GB drive's power cord.

6. Connect the two workstations with the 10Base-T (twisted-pair) cable into the RJ-11 (telephone-like) connectors labelled "TP".

7. Plug in the monitor and attach its cable to `larynx`'s monitor port. Since you'll be switching this shortly, don't bother screwing it in fully.

C.4 Workstation Boot Procedure

Now you're ready to start the machines. The following steps **must** be performed in the order given.

1. Turn on the powerbar. Hopefully nothing will start yet.

2. Turn on the external drives and let them spin up.

3. Turn on the monitor.

4. Turn on `larynx`.

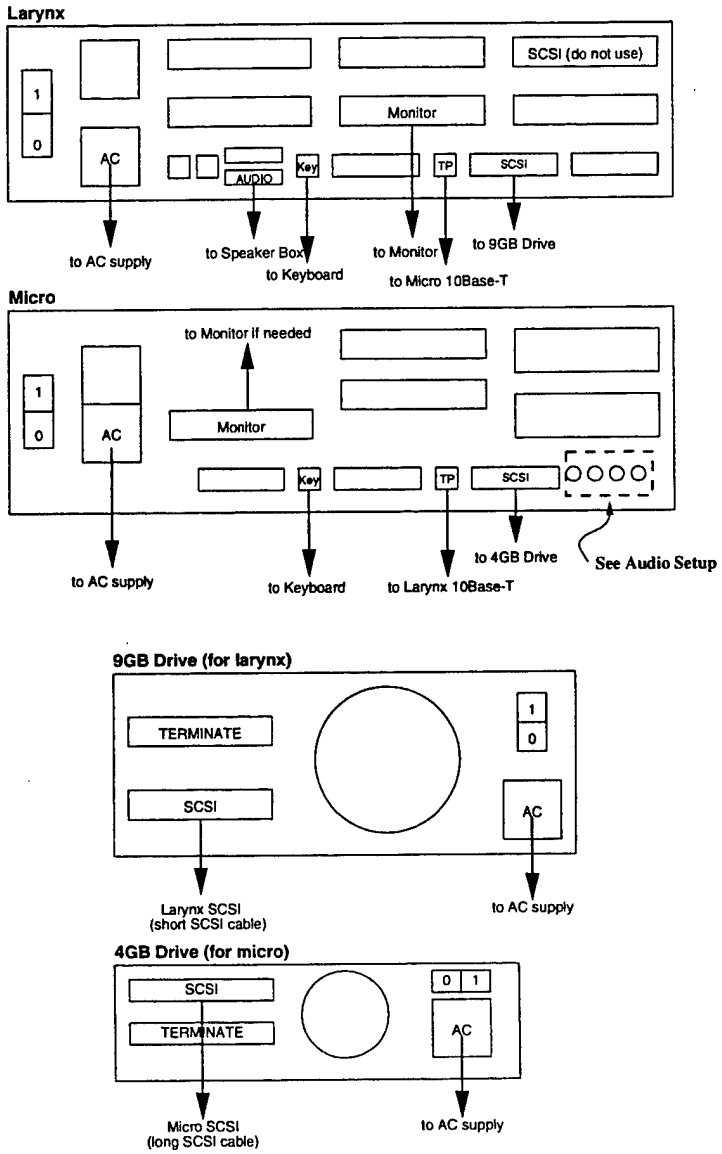

Figure 11: Travel system connection diagram

5. Let it boot to the point where it starts complaining about lack on carrier on `le0` or something like that. can't find its server on the network.

6. Interrupt with `L1-a` (*i.e.*, hold down the `Stop` key and type "a"), then type

   ```
   boot disk2
   ```

   at the PROM monitor prompt. Make sure you're using the right keyboard!

7. It should boot again, this time all the way to a login prompt. **Note:** You may get one or two messages about "hub link test disabled." Ignore them (they're just saying there's no one at the other end of the twisted-pair ethernet).

8. Now move the monitor connector to `micro`'s monitor port, and turn `micro` on. At some point (wherever it seems to be stuck), interrupt with `L1-a`. Make sure you're using the right keyboard! As before, do

   ```
   boot disk2
   ```

   at the PROM monitor prompt.

9. It should boot happily to a login prompt. At this point, I recommend having whoever will be running the demo login on `micro`'s console before switching the monitor back to larynx. My suggestion would be user `trains`, with password `trains96`, and hitting `Control-c` before X Windows starts. If you don't do this step now, or if another user wants to run the system, you need to logout and have them login, either blind or switching the monitor temporarily.

10. Move the monitor back to `larynx`, and you're ready to go. As noted above, I recommend logging in as user `trains` with password `trains96`. If you let X Windows start, it will launch the system automatically from an xterm window. You might want to complete the audio setup described in the next section before logging into `larynx` and starting the system.

This completes the setup of the workstations. It remains to setup the audio input and output for the speech processing, as described in the next section.

C.5 Audio Setup

This section describes how to hook up the audio components to provide (1) speech input and monitoring, and (2) audio output to both speaker and headphones. Note that I have referred to either `micro`'s back panel or `larynx`'s speaker box. Depending on your choice of audio host, you use one or the other consistently. The standard configuration (described here) is to use `micro` as audio host.

Figure 12 shows a schematic layout of the various devices and connections. The following steps needn't be performed in any particular order.

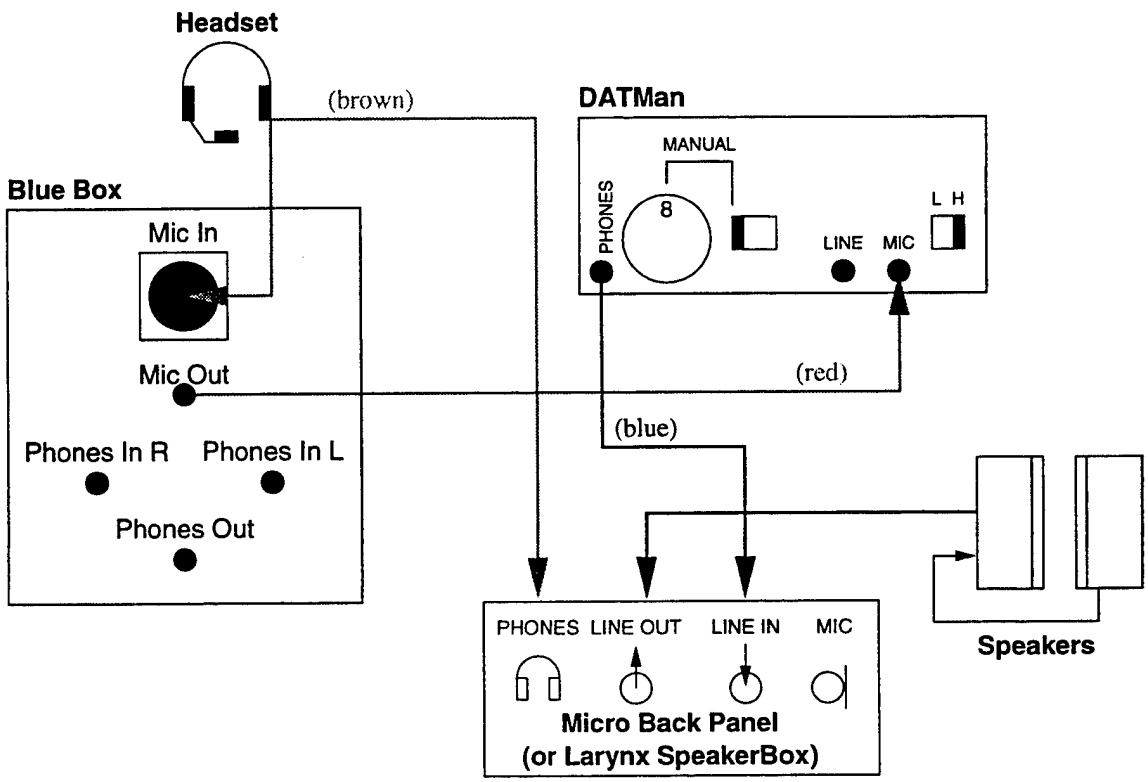

Figure 12: Travel system audio connection diagram

1. Connect the headset microphone to the blue box, using the large 3-pin plug into the "Mic In" socket. Leave the small plug on the headset (the headset phones) unconnected for now.

2. Connect the "Mic Out" on the blue box to the "Mic In" of the DATMan. This uses the red cable.

3. Connect the "Phones" jack of the DATMan to the "Line In" on `micro`'s back panel (or `larynx`'s speaker box) . "Line in" is the connector with the arrow pointing into the circle. This uses the blue cable.

4. The external speakers consist of a "main" speaker, with the volume and power controls, and a "slave" speaker without them. Plug the small grey cable connected to the main external speaker into the "Line Out" jack from `micro`'s back panel (or `larynx`'s speaker box). "Line out" is the connector with the arrow pointing out of the circle.

5. Connect the slave speaker to the back of the main speaker using the small grey cord attahced to the slave. It plugs into the "R Out" (or something) connector, in the back of the main speaker.

6. The DATMan and the external speaker need power through their adapters. I recommend using the extra powerbar to accomodate the adapters.

7. Plug the headphone connector from the headset (small plug on the same cord as the big 3-pin microphone plug) into the headphone jack on `micro`'s back panel (or `larynx`'s speakerbox). There may a problem with the length of that cord.

8. Finally, as a hack to allow the Help movie to be played through the external speakers, the "Line Out" from the **other** workstation (*i.e.*, `larynx` in this setup, or `micro` if you're using `larynx` as the audio host) needs to be connected to the "Input2" on the front of the master external speaker. **This should only be plugged into Input2 when you are playing the movie!** Otherwise it should be left unplugged since it disconnects the main speaker input (from the audio host). Since the Help movie is played through the speaker anyway, this may not even be necessary.

Now you need to setup the DATMan properly. Several of these settings can be experimented with, perhaps in conjunction with changing the record level using the Audio Manager on `larynx`.

1. A tape must be loaded, and the red "Rec" button must be pressed to enable the audio. You can adjust record levels at this point.

2. Volume should be set to about 16 using the "+" button on the front edge of the case (often hidden under the carrying case flap). **Note that the volume is reset whenever power is removed!**

3. "Mic Sens" should be set to "High". At least, I think so. It seemed to be set to "Low" at some point, "High" seems to work right.

4. "Rec Mode" set to "Manual" (I found it tended to cutoff and then be slow to readjust when set to "Auto/Speech").

5. "Rec Level" set to about 7; your mileage may vary. It seems to be okay if it clips a bit (on the DATMan VU meter).

Finally, I have set the defaults for the Audio Manager to work properly with DATMan settings as given above. You may have to adjust them further (either while running, or in the `trains` script for more permanent changes).

1. The Input Level is set to about 75, which gives good recognition when the DATMan volume is at 16.

2. The Output Level is set to about 80, which gives plenty of signal for the external speakers (which can always be turned down), but doesn't blast into the headphones too much. In a noisy room, this could go perhaps up (although watch for hiss and remember that the headphones and the line output are controlled together).

3. The Monitor Level should be set to give as much of the person's voice back into the headphones without putting so much through the speakers that you get feedback. I find it is very non-linear—it's barely noticeable and then suddenly feeds back. I guess it's not crucial that the person hear themselves, if push came to shove, and you could turn the Monitor Level to 0.

   Note: The latest TRAINS-96 Audio manager has no provision for adjusting the Monitor level. You could try using `/usr/demo/SOUND/gaintool` *or something...*

C.6 Troubleshooting

This section has a few tips about what can/might/will go wrong.

- Speech recognizer doesn't work: Make sure the DATMan is on and in "Rec" mode. If it is, make sure its headphone jack is connected to "Line In" and that the volume is cranked to about 16.

C.7 Shutdown

Shutdown is basically the reverse of setup. You need root privileges to do it nicely.

1. Rlogin to `micro` and do:

   ```
   % sux /etc/halt
   ```

   This will (obviously) close your remote login, and about 20 seconds later the system will be halted.

2. On `larynx`, the same command works:

```
% sux /etc/halt
```

You can then turn off both workstations and the monitor.

3. Turn off the external drives last.

If you don't have root privileges, just do a **sync** and switch off the power, in the order described above.

C.8 Shipping List

Figure 13 shows the packing of the cases. The following is a list of the components being shipped. Items marked with an asterisk are being shipped separately since they wouldn't fit in the cases.

```
1  Sun UltraSparc, serial no. 552F1388
1  Sun Sparc10, serial no. 249F5001
1  Sun 17 inch color monitor
1  9GB drive (for use with Sparc10)
1  4GB drive (for use with UltraSparc)
1  25--25 SCSI cable (for use with 4GB drive)
1  50--25 SCSI cable (for use with 9GB drive)
1  25-pin SCSI terminator (for use with 4GB drive)
1  50-pin SCSI terminator (for use with 9GB drive, two parts)
1  Sun audio/ethernet cable (for use with Sparc10, two parts)
1  Sun speaker box (for use with Sparc10)
2  Sun keyboards (one with cable for UltraSparc, one without for Sparc10)
2  Sun mice (optical for Sparc10, mechanical for UltraSparc)
1  Sun optical mouse pad
1  Sun keyboard cable (for Sparc10)
1  10Base-T (twisted-pair) crossover cable
5  A/C power cords (UltraSparc, Sparc10, 9GB drive, 4GB drive, monitor)
1  Large power bar
1  Small power bar
2  Locking security cables
2  External speakers (*)
1  DC adapter for speakers
1  Sony DATMan with carrying case (*)
1  DC adapter for DATMan
1  Magic blue box (tm)
1  Microphone headset (*)
5  Audio mini-jack patch cords (3 used, 2 spares)
2  Audio mini-jack splitters (spare)
```

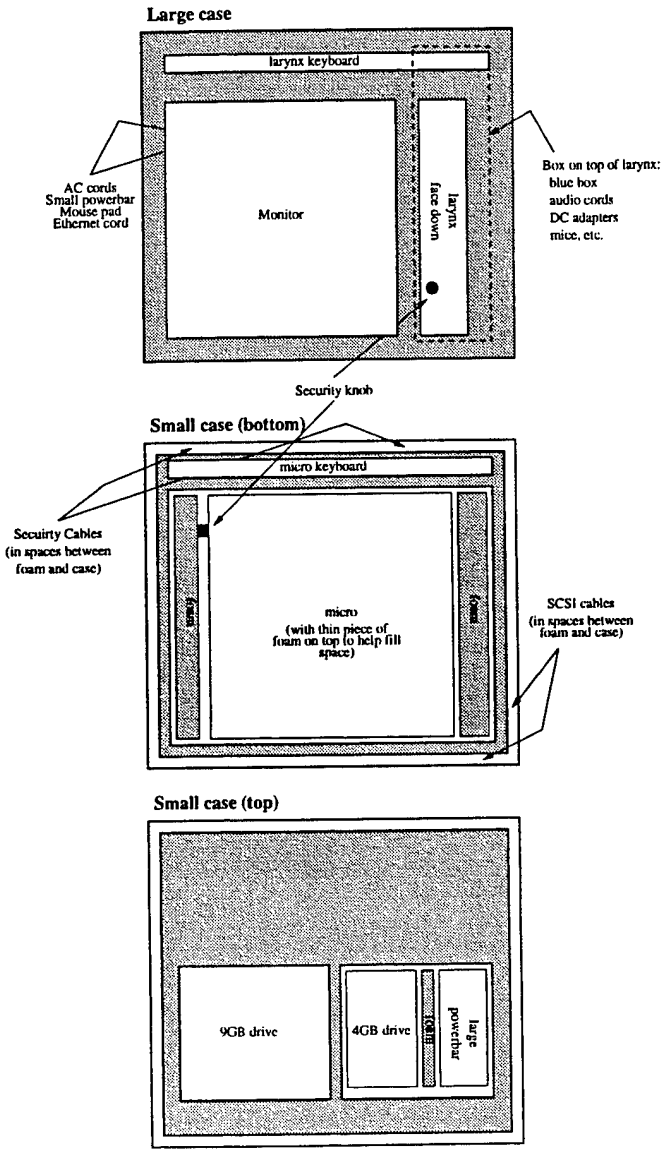

Figure 13: Travel system packing diagram

D Manual Pages

The following manual pages are provided in this section (as well as online):

`dlg_check`	Check TRAINS-96 dialogue contents
`dlg_org`	Organize TRAINS-96 dialogue contents
`dlg_play`	Play TRAINS-96 dialogue utterances
`taudio`	TRAINS Audio Manager
`tdisplay`	TRAINS Display Module
`tim`	TRAINS Input Manager
`tim_cat`	Send KQML messages from stdin to TRAINS IM
`tim_client`	Send and receive KQML messages to/from TRAINS IM
`tim_exec`	Exec a program with stdin/stdout connected to TRAINS IM
`tim_msg`	Send a KQML message to the TRAINS IM
`tkeyboard`	TRAINS Keyboard Manager
`tparser`	TRAINS Parser module
`tpm`	TRAINS Process Manager
`tpsm`	TRAINS Problem Solving module
`tpview`	TRAINS Parse Tree Viewer
`trains`	Run the TRAINS System
`treplay`	Replay a TRAINS System session
`tscenario`	TRAINS Scenario Chooser
`tsfx`	TRAINS Sound Effects module
`tshortcut`	TRAINS Shortcut Panel
`tspeech`	TRAINS version of Sphinx-II speech recognizer
`tspeechpp`	TRAINS Speech Post-Processor
`tspeechx`	TRAINS Speech Controller
`tsplash`	TRAINS Splash Screen module
`ttc`	TRAINS TrueTalk client using AudioFile server
`ttcl`	TRAINS Discourse Manager module
`ttranscript`	TRAINS Transcript module
`tts`	Runs TrueTalk server
`tttalk`	TRAINS Speech Generation module

NAME

dlg_check - Check TRAINS-96 dialogue contents

SYNOPSIS

dlg_check dir

DESCRIPTION

The dlg_check script checks the contents of the given dialogue directory recorded during a session with the TRAINS-96 system. If necessary, dlg_check first runs dlg_org(1) to organize the contents of the dialogue into suitable sub-directories.

OPTIONS

None.

USAGE

To check the contents of a dialogue, simply provide the path to the dialogue's log directory as an argument. For example:

```
% dlg_check /u/trains/96/2.0/logs/960821.1049
```

Relative paths work also.

ENVIRONMENT

TRAINS_BASE Used to find default startup file

FILES

TRAINS_BASE/bin/dlg_org is run if the dialogue directory needs to be organized into subdirectories.

SEE ALSO

dlg_org(1)

BUGS

This program must be modified if new logs or data files are created in the process of recording a TRAINS dialogue (i.e., if the contents of a log directory is changed).

AUTHOR

Eric Ringger (ringger@cs.rochester.edu)

NAME

dlg_org - Organize TRAINS-96 dialogue contents

SYNOPSIS

dlg_org dir

DESCRIPTION

The dlg_org script organizes the contents of the given dialogue directory into suitable sub-directories. The dialogue directory must have been recorded during a session with the TRAINS-96 system.

The following sub-directories are created:

data
: All audio (*.au) files are placed here.

mfc
: This sub-directory is initially empty, but MFC parameter files will be placed here.

s2
: This sub-directory is initially empty, but Sphinx-II log files will be placed here.

sent
: This sub-directory is initially empty, but hand transcriptions of the dialogue data will be placed here.

sys
: All system module logs and the transcript are placed here.

OPTIONS

None.

USAGE

To organise the contents of a dialogue, simply provide the path to the dialogue's log directory as an argument. For example:

```
% dlg_org /u/trains/96/2.0/logs/960821.1049
```

Relative paths work also.

ENVIRONMENT

TRAINS_BASE Used to find default startup file

SEE ALSO

dlg_check(1)

BUGS

This program must be modified if new logs or data files are created in the process of recording a TRAINS dialogue (i.e., if the contents of a log directory is changed).

AUTHOR

Eric Ringger (ringger@cs.rochester.edu)

NAME

dlg_play - Play TRAINS-96 dialogue utterances

SYNOPSIS

dlg_play dir [-silent]

DESCRIPTION

The dlg_play script plays the utterances in the given dialogue directory one at a time. The dialogue must have been recorded during a session with either the TRAINS-95 or the TRAINS-96 system.

The user is prompted before each utterance with several options: repeat, backup, quit, or continue. It is also possible to type n followed by a number to jump to the utterance in that position.

OPTIONS

-silent
: Indicates that the utterance audio should not actually be played. Used for testing.

USAGE

To play the utterances of a dialogue, simply provide the path to the dialogue's log directory as an argument. For example:

```
% dlg_org /u/trains/96/2.0/logs/960821.1049
```

Relative paths work also.

FILES

/s7/esps/bin/s16play from the ESPS software tool-set is used to play each utterance.

This Perl program uses the Curses module from the Net. Check CPAN for updates.

SEE ALSO

s16play(1)

BUGS

None known.

AUTHOR

Eric Ringger (ringger@cs.rochester.edu)

NAME

taudio - TRAINS Audio Manager

SYNOPSIS

taudio [-audio server] [-input N] [-output N] [-mic BOOL] [-linein BOOL] [-speaker BOOL] [-phones BOOL] [-lineout BOOL] [-meterRunning BOOL] [-debug where] [X args]

DESCRIPTION

Taudio is the TRAINS Audio Manager. It provides an X/Motif display for monitoring and adjusting audio input and output ports and levels.

OPTIONS

-audio host:device
: Connect to the AudioFile server at the given address. The default is the value of the environment variable AUDIOFILE, if set, otherwise the current host.

-input N
: Set initial input (record) level to N.

-output N
: Set initial output (play) level to N.

-mic BOOL
: Enable (True) or disable (False) the microphone input initially.

-linein BOOL
: Enable (True) or disable (False) the line level input initially.

-speaker BOOL
: Enable (True) or disable (False) the speaker output initially.

-phones BOOL
: Enable (True) or disable (False) the headphone output initially.

-lineout BOOL
: Enable (True) or disable (False) the line level output initially.

-meterRunning BOOL
: If True, the input level "VU meter" is enabled initially. The default is False, which reduces the amount of work the Audio Manager has to do.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args

Taudio accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Taudio first connects to the AudioFile server given by the -server argument or the AUDIOFILE environment variable. If any of the level options (e.g., -input) or port configuration options (e.g., -mic) were given, taudio configures the server, otherwise the settings are left unchanged.

Input and output levels can then be adjusted using the sliders, and input and output ports can be selected using the menus. Note that input ports are mutually exclusive, while output ports can all be on at once. The "VU meter" input level display can be enabled and disabled from the Input menu. It is recommended that it be left disabled once levels are set to reduce the load on the audio server.

AUDIO MANAGER MESSAGES

The following KQML messages are understood by the Audio Manager. They should be addressed with ":receiver AUDIO". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (exit :status N))
Requests that the Audio Manager exit with status N (default 0).

(request :content (hide-window))
Causes the Audio Manager display to iconify itself.

(request :content (show-window))
Causes the Audio Manager display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Ignored.

(request :content (end-conversation)
Ignored.

(request :content (chdir DIR))
Ignored.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
AUDIOFILE	HOST:DEVICE for AudioFile server

FILES

None.

DIAGNOSTICS

Some.

SEE ALSO

trains(1), tim(1)

BUGS

Not really a bug, but it would be nice to have some way to control the "monitor" level provided by the Sun audio hardware.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tdisplay - TRAINS Display Module

SYNOPSIS

```
tdisplay [-map STR] [-showMenus BOOL] [-showTextIn BOOL]
    [-showTextOut BOOL] [-textInHeight N] [-textOutHeight N]
    [-debug where] [X args]
```

DESCRIPTION

Tdisplay is the TRAINS Display module. It provides an object-oriented X window map display used by the generation component of the Discourse Manager. Objects on the map can be selected using the mouse, resulting in messages interpreted by the TRAINS Parser. The current version of the display also provides a "text out" window showing System utterances and a "text in" window into which the user can type (although the latter has been superseded by the Keyboard Manager, tkeyboard(1)).

OPTIONS

-map STR
: Specify the initial map displayed by the Display. Maps are looked for first in the current directory, then in TRAINS_BASE/etc/maps, both with and without the extension ".map". The default map is of the full-size Northeast U.S.

-showMenus BOOL
: Enable (True) or disable (False) the display of the application menubar. The default is True. Disabling it can be useful when screen real estate is at a premium.

-showTextIn BOOL
: Enable (True) or disable (False) the user input area underneath the map display.

-showTextOut BOOL
: Enable (True) or disable (False) the system output area above the map display.

-textInHeight N
: Specify the height of the user input area in pixels. The default is 100 (about five lines).

-textOutHeight N
: Specify the height of the system output area in pixels. The default is 20 (about one line).

-debug where

Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args

Tdisplay accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Tdisplay begins by reading its initial mapfile. Mapfiles are simply files containing the same KQML messages understood by the Display during normal operation. It then processes KQML messages on its standard input and outputs messages corresponding to user mouse actions to standard output (as selective broadcasts). The key to the Display module is the large number of display operations it can perform, as detailed in the next section.

DISPLAY MESSAGES

The following KQML messages are understood by the Display. They should be addressed with ":receiver DISPLAY". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

Object Manipulation Messages

The following messages create and otherwise manipulate Display objects.

```
(request :content (create :name :type :displayed :depth :bg
                          :color :fill :fillcolor :thickness :shape
                          [type-specific attrs]))
```

This creates a new Display object and is by far the most complicated request understood by the Display. That makes it as good a place as any to start.

The :name attribute names the new object. The :type attribute is one of city, track, route, engine, or region. The :displayed attribute can be t or nil, indicating whether the object is visible (and selectable) or not. The :depth attribute sets the depth of the object on the display--deeper objects (greater :depth) are drawn under less deep ones. The :bg attribute indicates whether the object is considered part of the "background" (i.e., is dynamic or not). See the SETBG request, below, for details. The

:color and :fillcolor attributes are X color names (or X RGB color specs); see X(1) for details. The :fill attribute varies from 0 (unfilled) to 100 (completely filled), and the :thickness attribute affects the borders of the object.

The shape of the object can be specified using the :shape attribute using one of the following forms:

```
(line :start LOC :end LOC)
(circle :center LOC :radius N)
(poylgon :center LOC :points|:rpoints (LOC1 LOC2 ...))
(multiline :points|:rpoints (LOC1 LOC2 ...))
```

The LOC attributes (locations), can be the names of objects, meaning their centers, or a list of (X Y) coordinates in the Display window (the origin is the upper, left corner). For polygon and multiline shapes, :points means the locations are absolute coordinates and :rpoints means they are relative (after the first one, of course). Note that most objects have a default shape.

Finally, the create request can specify attributes specific to the type of object being created. These additional attributes are:

```
city   :label STR :orientation O :ptsize N
track  :start LOC :end LOC
engine :at LOC :orientation O :outlined t|nil
route  :start CITY :tracks (OBJ1 OBJ2 ...)
```

The :orientation O is one of north, northeast, east, etc., describing the position of the label relative to the object (for cities) and the position of the engine relative to the city it is :at (for engines). The :outlined attribute for engines is an alternative for ":fill 0" or ":fill 100". The default shape for a track is a line connecting the endpoints, for an engine it is a simple schematic engine thingo, and for a route it is a series of spline curves along the tracks in the route.

(request :content (destroy OBJ))
: Destroys the named object, removing it from the Display.

(request :content (display OBJ))
: Causes OBJ to be displayed.

(request :content (undisplay OBJ))
: Causes OBJ to be neither seen nor selectable.

(request :content (set OBJ attr-value pairs))
: Sets attributes of the given OBJ. Not all attributes can be set in this way, but most, like :displayed and :color can be. The shape cannot be changed, however.

(request :content (default attr-value pairs))
Sets default values of some attributes for subsequent create requests. Again, most of the useful attributes can be set in this way, but not all of them.

Object Highlighting Messages

The following messages are used for the important job of highlighting object during a conversation.

(request :content (highlight OBJ :color :type :flash))
Causes the given object to be highlighted. The :type attribute can be object, circle, or rectangle, meaning that either the object itself becomes colored (for an object highlight) or that the appropriate colored shape is drawn around the object (actually around its bounding box). The :flash attribute can be nil, meaning don't flash (the default), t, meaning flash the highlight forever, or a number, meaning that the highlight should flash that many times and then unhighlight. Multiple highlights can be applied to an object and they are rendered in the order they were applied.

(request :content (unhighlight OBJ :color :type :flash))
Removes the matching highlight from the given object. If no attributes are given, all highlights are removed from the object.

Dialog Box Messages

The following messages provide popup dialog boxes of two types.

(request :content (confirm TAG STR))
Displays a blocking confirmer with STR as its text. When the user selects either OK or CANCEL, the Display outputs a reply with :content

(confirm TAG t|nil)

(request :content (dialog TYPE STR))
Displays a non-blocking dialog box of the given TYPE displaying STR. Currently only the :goals type is supported, and the string is displayed in a dialog box labelled "Goals for this TRAINS scenario". No output is generated if this dialog box is dismissed.

Display Control Messages

The following messages affect global properties of the Display.

(request :content (canvas :title STR :width N :height N))
Sets the title, height and width of the Display window. Changing

this other than at the start of a map will cause unpredictable results at best.

(request :content (translate X Y))
Translates the coordinates of subsequent request by the given amounts (in pixels). This affects primarily the interpretation of (X Y) pairs in create requests.

(request :content (scale X Y))
Scales the coordinates of subsequent request by the given amounts (floating point values). This affects primarily the interpretation of (X Y) pairs in create requests.

(request :content (setbg))
Sets the background pixmap of the Display's window to include any objects with the :bg attribute set to T. These objects are then not redrawn during Display updates. This is typically used once per map after the map objects have been created but before any engines, routes, etc. have been created.

(request :content (say STR))
Adds STR to the system output window above the map display.

(request :content (postscript FILENAME))
Dumps a color encapsulated Postscript file describing the current display to the given FILENAME.

(request :content (map FILENAME))
Causes the given mapfile to be read.

(request :content (refresh))
Causes the Display to redraw the map display (although see above regarding the setbg request).

(request :content (restart))
Causes the Display to erase all objects and re-read its original mapfile.

Module Control Messages

The following messages are the standard TRAINS System messages.

(request :content (exit :status N))
Requests that the Display exit with status N (default 0).

(request :content (hide-window))
Causes the Display to iconify itself.

(request :content (show-window))
Causes the Display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Treated as a RESTART followed by SHOW-WINDOW.

(request :content (end-conversation)
Treated as a HIDE-WINDOW.

(request :content (chdir DIR))
Ignored.

Display Output Messages

The following messages are generated by tdisplay as selective broadcasts in response to user keyboard and mouse actions.

(tell :content (mouse :select obj1 obj2 ...))
Sent when the user clicks on an object. The objects are all those within a certain fuzz factor of the click, ordered by depth.

(tell :content (mouse :drag obj :from obj :to obj1 obj2 ...))
Sent when the user drags an object. The destination object list is as described above for clicks.

(tell :content (confirm TAG t|nil))
Sent when the user answers a dialog box confirmer.

(tell :content (word W :index (I1 I2)))
Broadcast to announce a new word in the user's typed input. The :index argument identifies the start and end position of the word, which can in fact be several words as far as the parser is concerned, as in the token ''COULDN'T''. Index positions start at 1. A single number I can be given, implying ''(I I+1)''.

(tell :content (backto :index I))
Broadcast to indicate that any words previously output at index I or beyond (inclusive) were erased by the user.

(tell :content (end))
Broadcast to announce that the user hit Return.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_BASE	Used to find mapfiles

FILES

TRAINS_BASE/etc/maps	Default location of mapfiles

DIAGNOSTICS

Colormap complaints are possible, even common, if your colormap fills up. They should break anything though...

SEE ALSO

trains(1), tparser(1), ttcl(1)

BUGS

This code was due to be replaced long ago. Perhaps some day...

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tim - TRAINS Input Manager

SYNOPSIS

tim [-port N] [-nolog] [-showlogo BOOL] [-debug where]

DESCRIPTION

Tim is the TRAINS Input Manager. It accepts connections on a "well-known" socket (currently port 6200) and routes KQML messages between connected clients. It supports both true and selective broadcasts and provides full KQML syntax checking. The Input Manager also provides an optional graphical display of connected clients and the message traffic between them.

The Input Manager writes a log of all messages received and sent to the file "im.log" in the current directory (unless -nolog is given, see below). The chdir message (see below) causes the log to be closed and reopened in a new directory.

OPTIONS

-port N
: Listen on port N rather than the default (currently 6200 unless the environment variable TRAINS_SOCKET is set). Clients should connect to this port on the host running tim. If this port cannot be allocated, tim will increment the port number and keep trying until an available socket is found. Clients should follow the same protocol, calling connect(3) with successively higher port numbers.

-nolog
: If given, causes the Input Manager to not create the "im.log" file in the log directory.

-showlogo BOOL
: If True, the Input Manager starts by displaying the TRAINS logo. If False (the default), the clients and message traffic are displayed (see USAGE, below).

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

From the shell, usage is trivial: simply start tim and arrange for clients to connect to it. Normally this will be done by a script that launches the TRAINS system as a whole, and clients will be launched by the TRAINS process manager, tpm(1).

The Input Manager provides two different X/Motif displays. The first is a simple TRAINS Project logo, suitable for use in demos. The other, more useful but potentially confusing, display shows connected clients, their status, and message traffic between them. Clicking in the Input Manager's window toggles between the two types of display (the logo is obviously less work for the IM).

CLIENT USAGE

From a client application's perspective, using the Input Manager can be broken down into three steps:

1. Connect to tim on the well-known socket. See the discussion of the -port option, above, for details of this, and see any Unix IPC description for how to connect to a socket in general.
2. Register the name of the client with the Input Manager. To do this, the following KQML message must be sent (to the connected socket):

   (register :receiver im :name myname)

   An optional :class argument can be given to specify to which class a client belongs. Once this is done, messages sent to myname by other clients will appear on the socket.
3. Use the socket for communication with the Input Manager and other clients. A complete spec of the messages understood by the Input Manager is in the following section.

Messages not addressed to the Input Manager (IM) itself are copied to the receiver's connection (or an error is generated if the intended receiver does not exist). Messages without an explicit :receiver are considered broadcasts, and are sent to any module that has sent an appropriate LISTEN request to the Input Manager (selective broadcast, note the difference from the BROADCAST performative).

INPUT MANAGER MESSAGES

The following KQML messages are understood by the Input Manager. They should be addressed with ":receiver IM". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (define-class C :parent P))
Requests that the Input Manager define a new class named C whose parent class is P (default is the pre-defined class Any).

(register :name M :class C)
Sender is asking to be identified as M from now on. This should be the first message sent by any client. The client's status is set to CONNECTED after receipt of this message. The class C is optional and defaults to Any. Multiple REGISTER messages per client are permitted, and will result in messages for any of the registered names being sent to the client.

(tell :content (ready))
Sender is announcing that it is ''ready,'' whatever exactly that is taken to mean. The client's status is set to READY after receipt of this message.

(request :content (listen M))
Sender is asking to receive broadcast messages from module or class M. That is, messages sent by M (or by a module in class M) without an explicit :receiver will be copied to the sender of this message.

(request :content (unlisten M))
Sender is asking to stop receiving broadcast messages from module or class M.

(evaluate :content (status M))
Sender is asking for the status of module or class M. If M is a module, the response will be of the form:

```
(reply :sender IM :content (status M STATUS))
```

where the status is one of DEAD, CONNECTED, READY, or EOF. Any :reply-with in the original message will be used in the reply. If M is a class, one such reply will be sent for each module in the class. There is currently no mechanism for indicating that the replies are finished (we could do something smarter if this was needed).

(monitor :content (status M))
Sender is asking to be informed whenever the status of module or class M changes. This will result in REPLY messages as above, and again any :reply_with in the original MONITOR will be used in the REPLY. The Input Manager will send an initial REPLY with the current status of the module(s) immediately.

(unmonitor :content (status M))
Sender is requesting to stop monitoring status.

(request :content (chdir DIR))
Request that the Input Manager close the current "im.log" and open a new one in the given DIR.

(request :content (dump))
Request that the Input Manager dump its client table to stderr, for debugging purposes.

(request :content (exit :status N))
Request that the Input Manager exit (with optional status N). This closes all client connections, and so is an effective way to halt the entire system.

(broadcast :content PERFORMATIVE)
Causes the given PERFORMATIVE to be sent to all connected modules. Note that this is a true broadcast, as opposed to the selective broadcast provided by the LISTEN and UNLISTEN requests.

(request :content (hide-window))
Causes the Input Manager display to iconify itself.

(request :content (show-window))
Causes the Input Manager display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Ignored.

(request :content (end-conversation)
Ignored.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_SOCKET	HOST:PORT for Input Manager connection

FILES

im.log Input Manager log

DIAGNOSTICS

Copious.

In particular, when non-KQML input is received from a client (as can happen when it inadvertently prints an error message to its standard output, for example), the Input Manager complains to stderr and logs the bad input. This can be somewhat verbose due to the incremental parsing of KQML messages. Sorry.

SEE ALSO

tpm(1), tim_client(1)

BUGS

Undoubtedly.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tim_cat - Send KQML messages from stdin to the TRAINS Input Manager

SYNOPSIS

tim_cat [-socket HOST:PORT]

DESCRIPTION

Tim_cat is a simple program that reads KQML messages from its standard input and sends them to the TRAINS Input Manager.

OPTIONS

-socket HOST:PORT
: Connect to Input Manager at the given HOST and PORT. The default is to use port 6200 on the local host, or the value of the environment variable TRAINS_SOCKET, if set. Tim_cat will scan successive port numbers from that given trying to connect (see tim(1) for details).

ENVIRONMENT

TRAINS_SOCKET HOST:PORT at which to contact IM

FILES

None.

DIAGNOSTICS

None.

SEE ALSO

tim(1), tim_msg(1), tim_client(1)

BUGS

Probably not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tim_client - Send and receive KQML messages to/from TRAINS Input Manager

SYNOPSIS

tim_client [-socket HOST:PORT]

DESCRIPTION

Tim_client is a fairly simple program that reads KQML messages from its standard input and sends them to the TRAINS Input Manager, and reads messages from the Input Manager and prints them to its standard output.

OPTIONS

-socket HOST:PORT
: Connect to Input Manager at the given HOST and PORT. The default is to use port 6200 on the local host, or the value of the environment variable TRAINS_SOCKET, if set. Tim_client will scan successive port numbers from that given trying to connect (see tim(1) for details).

ENVIRONMENT

TRAINS_SOCKET HOST:PORT at which to contact IM

FILES

None.

DIAGNOSTICS

None.

SEE ALSO

tim(1), tim_msg(1), tim_cat(1)

BUGS

Probably not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tim_exec - Exec a program with stdin/stdout connected to TRAINS Input Manager

SYNOPSIS

tim_exec [-socket HOST:PORT] cmd [args]

DESCRIPTION

tim_exec is a simple program that connects to the TRAINS Input Manager, then uses the rest of its command-line arguments as a command to launch with stdin and stdout connected to the IM.

OPTIONS

-socket HOST:PORT
: Connect to Input Manager at the given HOST and PORT. The default is to use port 6200 on the local host, or the value of the environment variable TRAINS_SOCKET, if set. Tim_exec will scan successive port numbers from that given trying to connect (see tim(1) for details).

ENVIRONMENT

TRAINS_SOCKET HOST:PORT at which to contact IM

FILES

None.

DIAGNOSTICS

None.

SEE ALSO

tim(1), tim_msg(1), tim_cat(1), tim_client(1)

BUGS

Probably not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tim_msg - Send a KQML message to the TRAINS Input Manager

SYNOPSIS

tim_msg [-socket HOST:PORT] verb [parameters]

DESCRIPTION

Tim_msg is a simple program that allows a single KQML message to be sent to the TRAINS Input Manager. It first connects to the Input Manager (see the -socket option, below). The arguments are then formatted as a KQML performative and sent over the connection. Note that the parameters are not checked for KQML syntactic correctness. Be sure to escape parentheses, spaces, and the like from the shell.

OPTIONS

-socket HOST:PORT
: Connect to Input Manager at the given HOST and PORT. The default is to use port 6200 on the local host, or the value of the environment variable TRAINS_SOCKET, if set. Tim_msg will scan successive port numbers from that given trying to connect (see tim(1) for details).

USAGE

This command tells the IM to terminate, thereby terminating all the other modules:

```
tim_msg request :receiver im :content '(exit)'
```

This simple example simulates a word being recognized by Sphinx:

```
tim_msg tell :sender speech-in :content '(word "hello" :index 1)'
```

ENVIRONMENT

TRAINS_SOCKET HOST:PORT at which to contact IM

FILES

None.

DIAGNOSTICS

None.

SEE ALSO

tim(1), tim_cat(1), tim_client(1)

BUGS

Probably not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tkeyboard - TRAINS Keyboard Manager

SYNOPSIS

tkeyboard [-rows N] [-columns N] [-bufnum N] [-grab BOOL]
[-hotkey STR] [-showMenus BOOL] [-fontpat STR] [-fontsize N]
[-debug where] [X args]

DESCRIPTION

Tkeyboard is the TRAINS Keyboard Manager. It provides an X/Motif window into which the user can type. Typed input is incrementally output to standard output as KQML messages. It also provides a "hotkey" to allow speech recognition to be stopped and started using the keyboard, thereby freeing up the mouse for concurrent graphical gestures.

OPTIONS

-rows N
: Number of rows for Keyboard Manager window (default 7).

-columns N
: Number of columns for Keyboard Manager window (default 80).

-bufnum N
: Specify that the last N words of the user's input should not be output (until Return is typed). This allows the user to make corrections locally without the need for a BACKTO message. The default is 2.

-grab BOOL
: Enable (True) or disable (False) the initial keyboard grab. The default is True. See USAGE below for more details.

-hotkey STR
: Set the "hotkey" used to control speech recognition. The STR should be the name of an X11 KeySym, such as "a" (not a good choice, however) or "Alt_L" (the default). Be careful when changing this since most keys are either used in typed input and/or are auto-repeating.

-showMenus BOOL
: Enable (True) or disable (False) the display of the application menubar. The default is True. Disabling it can be useful when screen real estate is at a premium.

-fontpat STR
: Set the font pattern to the given STR, which should have a single "default is a pattern representing Courier medium.

-fontsize N
: Set the initial font size to N. The default is 14.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args
: Tkeyboard accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Tkeyboard incrementally processes typed input and outputs words in a format similar to the speech recognition modules tspeechin(1) and tspeechpp(1).

Unless disabled with "-grab False", tkeyboard places an active grab on the user's keyboard, thereby allowing it to receive keyboard events regardless of the location of the mouse pointer. It automatically releases the grab when iconified, and re-grabs when de-iconified. The Control menu provides items to grab or ungrab the keyboard unconditionally.

In conjunction with the keyboard grabbing, a "hotkey" is defined that, when pressed, generates a START request to the SPEECH-IN module and when released generates a STOP request. The default hotkey is the left Alt key, which is convenient since it does not auto-repeat and is not used for typed input. Use of the hotkey allows the mouse pointer to be used simultaneously in another window, for example the DISPLAY.

KEYBOARD MANAGER MESSAGES

The following KQML messages are understood by the Keyboard Manager. They should be addressed with ":receiver KEYBOARD". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (grab))
: Causes the Keyboard Manager to grab the user's keyboard.

(request :content (ungrab))
: Causes the Keyboard Manager to release its grab on the user's keyboard.

(request :content (reset))
: Clears the Keyboard Manager display and resets any word buffering.

(request :content (exit :status N))
Requests that the Keyboard Manager exit with status N (default 0).

(request :content (hide-window))
Causes the Keyboard Manager display to iconify itself.

(request :content (show-window))
Causes the Keyboard Manager display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Treated as a RESET followed by SHOW-WINDOW.

(request :content (end-conversation)
Treated as a HIDE-WINDOW.

(request :content (chdir DIR))
Ignored.

The following messages are generated by tkeyboard as selective broadcasts during recognition.

(tell :content (word W :index (I1 I2)))
Broadcast to announce a new word in the user's input. The :index argument identifies the start and end position of the word, which can in fact be several words as far as the parser is concerned, as in the token ''COULDN'T''. Index positions start at 1. A single number I can be given, implying ''(I I+1)''.

(tell :content (backto :index I))
Broadcast to indicate that any words previously output at index I or beyond (inclusive) were erased by the user.

(tell :content (end))
Broadcast to announce that the user hit Return.

ENVIRONMENT

DISPLAY HOST:SCREEN for X server

FILES

None.

DIAGNOSTICS

Sometimes complains that a grab couldn't be obtained if, for example, another application already had a grab on the keyboard (e.g., menus). The whole grab thing is complicated.

SEE ALSO

trains(1), tim(1)

BUGS

Grabs are annoying.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tparser - TRAINS Parser module

SYNOPSIS

tparser [lisp args]

DESCRIPTION

Tparser is a dumped Allegro Common Lisp image that implements the TRAINS Parser module, a robust, bottom-up chart parser. It reads KQML messages (describing input words) from standard input and writes KQML messages (describing utterance interpretations) to standard output.

OPTIONS

You can pass any arguments suitable for dumped Lisp images. I don't know why you would want to, however.

USAGE

Tparser asks the Input Manager to LISTEN to the USER-INPUT class of modules. This class includes the speech recognition modules SPEECH-IN and SPEECH-PP, the Keyboard Manager KEYBOARD, and the DISPLAY. As these modules broadcast the user's spoken, typed, or graphical input, the parser constructs a representation of the meaning of the utterance. When the utterance is complete, it broadcasts a TELL message whose content is the interpretation.

Currently, although the Parser listens to both SPEECH-IN and SPEECH-PP messages, it only uses one of them. Its policy is to use all of SPEECH-PP's output if that module output anything this utterance, otherwise use SPEECH-IN's. This can cause problems if, for example, SPEECH-IN completely finishes processing before SPEECH-PP even outputs START.

PARSER MESSAGES

The following KQML messages are understood by the Parser. They should be addressed with ":receiver PARSER". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(tell :content (start :uttnum N))
: Causes the parser to start processing a new utterance.

(tell :content (word W :uttnum N :index I :frame F :score S))
: Adds a word to the current utterance. The format is more fully described in the manpages for tspeech(1), tspeechpp(1), and tkeyboard(1).

(tell :content (backto :uttnum N :index I))
Invalidates words at index positions I and higher (inclusive).

(tell :content (end :uttnum N))
Causes the parser to stop processing the utterance and output its interpretation.

(tell :content (mouse :select obj1 obj2 ...))
Sent by DISPLAY when the user clicks on an object.

(tell :content (mouse :drag obj :from obj :to obj1 obj2 ...))
Sent by DISPLAY when the user drags an object.

(tell :content (confirm TAG t|nil))
Sent by DISPLAY when the user answers a dialog box confirmer.

(request :content (restart))
Clears any internal state in the parser.

(request :content (parse-tree))
Causes the Parser to reply with a message containing the latest parse tree. This is used by the Parse Tree Viewer, tpview(1).

(request :content (exit :status N))
Requests that the Parser exit with status N (default 0).

(request :content (chdir DIR))
Causes the Parser to close its "parser.log" file and reopen it in the given DIR.

(request :content (start-conversation :name N :lang L :sex S))
Logged but otherwise ignored.

(request :content (end-conversation)
Logged but otherwise ignored.

The following message is generated by tspeech as a selective broadcast at the end of each utterance:

(tell :content (SPEECH-ACT PARMS))
PARMS include: :objects, :paths, :semantics, :noise, :social-context, :relaibility, :mode, :syntax, :setting, and :input. A more complete description of the Parser output can be found in its online documentation and various technical reports.

ENVIRONMENT

The dumped Allegro CL image depends on a shared library for execution. You can specify the location of this library using the environment variable ALLEGRO_CL_HOME if it is not in the same place as when the image was dumped.

FILES

parser.log Parser log

DIAGNOSTICS

Complains about unknown words.

SEE ALSO

trains(1)

BUGS

Doesn't seem to crash too often anymore.

AUTHOR

James Allen (james@cs.rochester.edu).

NAME

tpm - TRAINS Process Manager

SYNOPSIS

tpm [-socket host:port] [-debug where]

DESCRIPTION

Tpm is the TRAINS Process Manager. It provides process management services in response to KQML messages. It can launch processes (optionally connecting them to the TRAINS Input Manager), as well as killing them. Note that process status reporting is now handled by the Input Manager.

OPTIONS

-socket host:port
: Connect to the Input Manager on host at port rather than the default (localhost:6200, unless the environment variable TRAINS_SOCKET is set). If the Input Manager cannot be found, processes can still be launched but will be unable to exchange messages (which sort of limits their use, I would think). The Process Manager will scan successive ports attempting to connect to the Input Manager if the initial value is unconnectable.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

After starting tpm from the shell (which has to happen after the Input Manager, tim(1), is started), the TRAINS system is typically bootstrapped by sending initial messages to the Process Manager using one of the Input Manager utilities like tim_cat(1).

PROCESS MANAGER MESSAGES

The following KQML messages are understood by the Process Manager. They should be addressed with ":receiver PM". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (start :name M :host HOST :exec FILE
:argv (args) :envp (envs) :connect t|nil))

Sender is asking to start a process named M executing FILE on HOST (default is the local host). The argument list and environment of the process can be set with the optional parameters. Note that :argv, if used, must specify argv[0], typically the basename of the executable, and :envp, if used, should be a list of strings of the form "VAR=VALUE". The elements of :argv and :envp can be tokens or strings (and must be strings is they include whitespace and the like).

If the :connect parameter is non-nil (the default), the process is started with stdin and stdout connected to the Input Manager (if possible). Otherwise the process can find the address of the Input Manager in its environment as the value of TRAINS_SOCKET in the form "host:port" (if the Process Manager knows the address at connect time). It must then arrange to connect to the Input Manager by itself using this information.

(request :content (kill M :signal N))

Sender is asking to terminate process M by sending signal N (default 2, SIGTERM).

(request :content (dump))

Requests the Process Manager to dump a description of the current set of processes to stderr.

(request :content (exit :status N))

Requests that the Process Manager itself exit with status N (default 0). This terminates all processes managed by the Process Manager.

(request :content (chdir DIR))

Ignored.

(request :content (show-window))

Ignored.

(request :content (hide-window))

Ignored.

(request :content (start-conversation :name N :lang L :sex S))

Ignored.

(request :content (end-conversation)

Ignored.

ENVIRONMENT

TRAINS_SOCKET HOST:PORT for Input Manager connection

FILES

None.

DIAGNOSTICS

Copious.

In particular, when a process managed by the Process Manager dies, the PM will (usually) complain that "read was interrupted". This is normal: the PM was asleep in read(2) when the SIGCHLD for the child's death was received (and handled). I could specialcase the error message, but it doesn't seem worth it. The TRAINS processes aren't supposed to die anyway...

SEE ALSO

trains(1), tim(1), tim_client(1)

BUGS

Undoubtedly, although this program is now fairly simple.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tpsm - TRAINS Problem Solving module

SYNOPSIS

tpsm [lisp args]

DESCRIPTION

Tpsm is a dumped Allegro Common Lisp image that implements the TRAINS Problem Solving module. The PS module manages planning and plan recognition services for the TRAINS System. It reads KQML messages (requests) from standard input and writes KQML messages (replies) to standard output.

OPTIONS

You can pass any arguments suitable for dumped Lisp images. I don't know why you would want to, however.

USAGE

The Problem Solving module maintains a hierarchical goal tree as well as a linear history of the problem solving state. Requests perform operations on the goal tree and/or move back and forth through the history. An important aspect of the PS module is that it supports two different classes of requests:

1. Interpretation requests, which test the feasibility of an operation without actually recording it; and
2. Update requests, which modify the problem solving state.

This allows other modules to pursue alternative interpretations prior to committing to one.

PROBLEM SOLVING MESSAGES

I am not going to attempt to document the PS module here. Further information is available in online documentation and various technical reports.

ENVIRONMENT

The dumped Allegro CL image depends on a shared library for execution. You can specify the location of this library using the environment variable ALLEGRO_CL_HOME if it is not in the same place as when the image was dumped.

FILES

ps.log Problem Solving log

DIAGNOSTICS

Not many. Uses ERROR and SORRY performatives to indicate problems.

SEE ALSO

trains(1)

BUGS

Crashes regularly. On the other hand, its behavior has increased the robustness of the rest of the system that has to deal with it. At least it (usually) doesn't drop into the Lisp debugger.

AUTHOR

James Allen (james@cs.rochester.edu).

NAME

tpview - TRAINS Parse Tree Viewer

SYNOPSIS

tpview

DESCRIPTION

Tpview is the TRAINS Parse Tree Viewer. It provides a button that, when pressed, asks the TRAINS parser for a summary of the chart for the previous user utterance. This chart summary is used to draw the parse trees in an X/Tk display. Successive parse trees are appended to the display each time the user presses the button.

The Parse Tree Viewer is implemented using Perl version 5 and the Perl/Tk package.

OPTIONS

None.

PARSE TREE VIEWER MESSAGES

The following KQML messages are understood by the Parse Tree Viewer. They should be addressed with ":receiver PVIEW". Case is insignificant outside of strings. Whitespace between messages is ignored.

(tell :content (parse-tree P))

Sent by the Parser in response to tpview's request when the user presses the "Get Parse Tree" button. The parse tree P is a parenthesized list of non-terminal and terminal entries. Non-terminal entries have the form:

"NTX" NT 1 NTI 2 NTJ 3 NTK

where "NTX" is a quoted node-name consisting of a concatenated node-type and number, NT is the node-type for the preceding node-name, NTI is the first daughter (also a node-name) of the preceding node-name, and the sucessors are the second, third, etc. daughters of the preceding node-name. These daughter node-names are defined by subsequent entries in P.

Terminal entries have the form:

"NTX" NT LEX W

where "NTX" is again a quoted node-name consisting of a concatenated node-type and number, NT is again the node-type for the preceding node-name, and W is the word for the terminal node.

(request :content (show-window))
Requests that tpview de-iconify itself.

(request :content (hide-window))
Requests that tpview iconify itself.

(tell :content (start-conversation))
Ignored.

(tell :content (end-conversation))
Ignored.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_BASE	Used to find Perl libraries

FILES

TRAINS_BASE/etc/pview/kqml.perl	KQML Parsing routines.
TRAINS_BASE/etc/pview/tree.perl	Tree Drawing routines.

DIAGNOSTICS

A few warnings.

SEE ALSO

tparser(1)

BUGS

I expect so.

AUTHOR

Eric Ringger (ringger@cs.rochester.edu).

NAME

trains - Run the TRAINS System

SYNOPSIS

trains [version]

DESCRIPTION

The script trains runs the TRAINS system by performing the following functions:

1. Examining environment variables;
2. Starting the AudioFile server;
3. Starting the Input Manager, tim(1), and the Process Manager, tpm(1);
4. Starting other modules by sending START messages to the Process Manager using tim_cat(1).

The trains script is the place to start if you want to customize some aspect of the TRAINS System, such as running a different executable for a particular module, or running a different set of modules altogether.

OPTIONS

None.

USAGE

You simply run trains from your shell. It will even check that X is running and, if it isn't, will start the X server with a configuration suitable for demos. How great is that?

ENVIRONMENT

The following environment variables are checked by the trains script.

DISPLAY
: X Windows display. The default is screen 0 on the local host.

TRAINS_BASE
: Root of TRAINS directory tree. The default is currently "/u/trains/96/2.0".

TRAINS_LOGS
: Directory for session logs. The default is "TRAINS_BASE/logs".

TRAINS_SPEECH_IN_HOST
: Host to run speech recognition modules tspeech(1) and tspeechpp(1). The default is "micro".

TRAINS_LISP_HOST
: Host to run Lisp modules ttcl(1), tparser(1), and tpsm(1). The default is "milli".

TRAINS_DM_HOST
: Host to run ttcl(1) (overrides TRAINS_LISP_HOST).

AUDIOFILE
: If set to HOST:0, AudioFile audio server runs on HOST.

TRAINS_AUDIO_HOST
: Host to run AudioFile audio server. The default is the same host as used for DISPLAY.

TRAINS_USER_SEX
: If set, for example to "f", this is passed as the -sex argument to tspeech(1).

FILES

None.

DIAGNOSTICS

None of its own.

SEE ALSO

tim(1), tpm(1)

BUGS

Unlikely.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

treplay - Replay a TRAINS System session

SYNOPSIS

treplay [-audio BOOL] logdir

DESCRIPTION

The script treplay is a variant of the basic trains script (see trains(1)) that starts the TRAINS System in replay mode. This involves running only the necessary interface components and none of the back-end reasoners. It also runs an additional module, REPLAY, that parses the "im.log" file in the given directory and plays back the messages in real-time. An X/Tk interface provides a display of the messages being replayed as well as "tapedeck-style" buttons for controlling the playback.

OPTIONS

-audio BOOL
: If True (the default), user and system audio is replayed. If False, it isn't, and the audio modules and AudioFile server are not started.

USAGE

You run treplay from your shell and tell it the name of the log directory for the session you want to replay. User utterances are replayed by playing the audio files recorded by the speech recognizer tspeech(1). System utterances are recreated using tttalk(1) (i.e., The TrueTalk speech generator). System displays are recreated using the Display module.

The Replay display provides a display of the messages parsed out of the im.log, as well as several buttons for controlling playback. The "Play" button starts real-time playback. The current dialogue time (from time 0 at the start of the dialogue) is shown in the upper left of the Replay display. During a delay between messages, the delay time counts down above the dialogue time. The "Play" button changes into a "Pause" button that, amazingly enough, will pause playback when pressed. Other buttons allow you to skip forward or back one message, one utterance, or to the start or end of the conversation.

Note: As I write this, stepping forward one message or utterance does not reset the delay countdown timer. You should pause and play (i.e., double-click the Play/Pause button) in order to skip a lengthy delay.

ENVIRONMENT

The following environment variables are checked by the treplay script.

DISPLAY
: X Windows display. The default is screen 0 on the local host.

TRAINS_BASE
: Root of TRAINS directory tree. The default is currently "/u/trains/96/2.0".

AUDIOFILE
: If set to HOST:0, AudioFile audio server runs on HOST.

TRAINS_AUDIO_HOST
: Host to run AudioFile audio server. The default is the same host as used for DISPLAY.

FILES

None.

DIAGNOSTICS

Probably not.

SEE ALSO

trains(1)

BUGS

The Perl script that implements the REPLAY module (i.e., that parses the "im.log" and sends the messages) can get confused pretty easily. For example, overlapping user speech utterances, where the user didn't wait long eneough between utterances, will confuse audio playback.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tscenario - TRAINS Scenario Chooser

SYNOPSIS

tscenario [-f FILE] [-debug where] [X Options]

DESCRIPTION

Tscenario is the TRAINS Scenario Chooser. It provides buttons for specifying either random or preset scenarios. It also arranges to step through preset scenarios one conversation at a time.

OPTIONS

-f FILE
: Specify startup file read to define the initial set of preset scenarios. The default is to read from TRAINS_BASE/etc/scenario.rc.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

Selecting a random scenario sends a message of the form

```
(request :receiver DM :content (config enum N))
```

to the Dialogue Manager, specifying N randomly-placed engines in the scenario.

Selecting a preset scenario sends the corresponding config message (as defined by the DEFINE message that defined the button, see below) to the Discourse Manager.

If a preset scenario is selected, then when the Scenario Chooser receives an END-CONVERSATION message, it automatically selects the next preset scenario (wrapping around at the end) and sends the appropriate message to the DM.

SCENARIO CHOOSER MESSAGES

The following KQML messages are understood by the Scenario Chooser. They should be addressed with ":receiver SCENARIO". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (define :label L :content C)
Creates a new preset scenario button labelled L which sends content C (typically a CONFIG message) to the DM when selected.

(tell :content (chdir DIR))
Ignored.

(tell :content (start-conversation))
Ignored.

(tell :content (end-conversation))
If a preset scenario is selected, receipt of this message causes the Scenario Chooser to switch to the next scenarion and send the appropriate message to the DM.

(request :content (hide-window))
Request that the Scenario Chooser iconify its window.

(request :content (show-window))
Request that the Scenario Chooser deiconify its window.

(request :content (exit :status N))
Request that the Scenario Chooser exit (with optional status N (default 0).

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_BASE	Used to find default startup file

FILES

TRAINS_BASE/etc/scenario.rc	Default startup file

DIAGNOSTICS

Maybe.

SEE ALSO

trains(1), ttcl(1)

BUGS

Possibly.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tsfx - TRAINS Sound Effects module

SYNOPSIS

tsfx [-audio server] [-debug where]

DESCRIPTION

Tsfx is the TRAINS Sound Effects module. In fact, all it does is play audio files using the AudioFile server in response to KQML requests. This is very useful for replay, however.

OPTIONS

-audio host:device
: Connect to the AudioFile server at the given address. The default is the value of the environment variable AUDIOFILE, if set, otherwise the current host.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

Tsfx first connects to the AudioFile server given by the -server argument or the AUDIOFILE environment variable. Then, in response to PLAY requests, it opens the specified audio file and sends it to the AudioFile server. The file is assumed to contain appropriate data, in this case 16 kHz, 16-bit, linear-encoded data with no headers. When the audio finishes playing, a DONE reply is generated.

Note that tsfx currently processes only a single PLAY request at a time. It would not be too hard to get it to play multiple files simultaneously. It would also be possible to have it recognize other audio file types and do the appropriate conversions.

SOUND EFFECTS MESSAGES

The following KQML messages are understood by the Audio Manager. They should be addressed with ":receiver SFX". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (play STR))
Plays the audio file named by STR. When audio output is complete, a DONE reply is generated.

(request :content (exit :status N))
Requests that the Audio Manager exit with status N (default 0).

(request :content (hide-window))
Ignored.

(request :content (show-window))
Ignored.

(request :content (start-conversation :name N :lang L :sex S))
Ignored.

(request :content (end-conversation)
Ignored.

(request :content (chdir DIR))
Ignored.

ENVIRONMENT

AUDIOFILE HOST:DEVICE for AudioFile server

FILES

None.

DIAGNOSTICS

Unlikely.

SEE ALSO

trains(1), taudio(1)

BUGS

Unlikely.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tshortcut - TRAINS Shortcut Panel

SYNOPSIS

tshortcut [-f FILE] [-debug where] [X Options]

DESCRIPTION

Tshortcut is the TRAINS Shortcut Panel. It allows arbitrary messages to be sent to modules of the TRAINS System by clicking on buttons. The set of buttons can be edited to add new buttons or modify existing ones.

OPTIONS

-f FILE
: Specify startup file read to define the initial set of shortcut definitions. The default is to read from TRAINS_BASE/etc/shortcut.rc.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

Double-clicking on a shortcut item sends the corresponding performative into the system via the Input Manager.

The File menu provides the following items for manipulating shortcuts: "New" allows the creation of a new shortcut, "Delete" deletes the currently-selected shortcut, "Edit" edits it, and "Save As" allows the current set of shortcuts to be saved to a file (for later use with the -f option).

SHORTCUT PANEL MESSAGES

The following KQML messages are understood by the Shortcut Panel. They should be addressed with ":receiver SHORTCUT". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (define :label L :content C)
: Creates a new shortcut labelled L which sends content C (a performative) when selected.

(request :content (hide-window))
Request that the Shortcut Panel iconify its window.

(request :content (show-window))
Request that the Shortcut Panel deiconify its window.

(request :content (exit :status N))
Request that the Shortcut Panel exit (with optional status N (default 0).

(tell :content (start-conversation))
Ignored.

(tell :content (end-conversation))
Ignored.

(tell :content (chdir DIR))
Ignored.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_BASE	Used to find default startup file

FILES

TRAINS_BASE/etc/shortcut.rc	Default startup file

DIAGNOSTICS

Maybe.

SEE ALSO

trains(1), tim(1)

BUGS

Possibly.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tspeech - TRAINS version of Sphinx-II speech recognizer

SYNOPSIS

tspeech [-audio server] [-bufnum N] [-sex m|f] [-dictfn file] [-lmfn file]

DESCRIPTION

Tspeech is a script that runs a version of the Sphinx-II speech recognition system from Carnegie Mellon University, modified to work as a component of the TRAINS system. This provides speaker-independent, incremental speech recognition of words from audio provided by an AudioFile server.

Tspeech is a "faceless" application, i.e., it has no display component itself. It responds to KQML messages on stdin that start and stop recognition, and broadcasts the results of the recognition to stdout. When connected to the Input Manager, tim(1), these outputs are treated as selective broadcasts and are sent to any client that has LISTENed for them. Tspeech also records the input audio in files named "utt.NNN.au", where NNN is an utterance counter, and records the results of the recognition in files named "utt.NNN.out". The audio files contain 16 kHz, 16-bit, linear-encoded data with no headers.

OPTIONS

-audio host:device
: Connect to the AudioFile server at the given address. The default is the value of the environment variable AUDIOFILE, if set, otherwise the current host.

-bufnum N
: Specify that the last N words of the current hypothesis should not be output (until the utterance is ended). This prevents constant revision at the "frontier" of the recognition, at the cost of some latency in the output. The default is 2.

-sex m|f
: Specify the sex of speaker, used to select an appropriate acoustic model. The default is 'm'.

-dictfn file
: Specify the phonetic dictionary to be used. The default is the TRAINS Dialogue Corpus dictionary, "tdc-75.dic". dictionary.

-lmfn file
: Specify the language model to be used. The default is the TRAINS Dialogue Corpus model, "tdc-75.bigram".

USAGE

Tspeech first loads its acoustic and language model files, then connects to the AudioFile server. It is assumed that the server is running using an audio format compatible with Sphinx-II, namely 16 kHz, 16-bit, linear-encoded samples.

Receipt of a START message initiates recognition, which includes incrementing the utterance counter and broadcasting a START message. Words are broadcast as they are recognized, with a BACKTO message being used to indicate revision of a word previously output. The last few words of the hypothesis are buffered to prevent excessive BACKTO's (see the -bufnum argument). Receipt of a STOP message terminates the utterance; tspeech then finishes processing the utterance, outputs any remaining words, and finally outputs an END message.

Note that the START and STOP messages are usually generated (in the TRAINS System, at least) by the tspeechx(1) Speech Controller. This program also provides a display of speech recognition results from both tspeech and the Speech Post-Processor, tspeechpp(1).

TSPEECH MESSAGES

The following KQML messages are understood by tspeech. They should be addressed with ":receiver SPEECH-IN". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (chdir DIR))
: Request that the Speech recognizer start writing its output files ("utt.NNN.au" and "utt.NNN.out") in the given DIR.

(request :content (start))
: Start recognition. Initiates processing of audio and broadcast of recognized words.

(request :content (stop))
: Stop recognition. Stops processing audio and completes processing of the current utterance.

(request :content (exit :status N))
: Request that tspeech exit with optional status N.

(request :content (start-conversation :name N :lang L :sex S))
: Resets utterance counter to 1.

(request :content (end-conversation)
Ignored.

(request :content (show-window))
Ignored.

(request :content (hide-window))
Ignored.

The following messages are generated by tspeech as selective broadcasts during recognition.

(tell :content (start :uttnum N))
Broadcast to announce the start of a new utterance.

(tell :content (word W :uttnum N :index (I1 I2) :frame (F1 F2)))
Broadcast to announce a new word in the hypothesis. The :index argument identifies the start and end position of the word, which can in fact be several words as far as the parser is concerned, as in the tokens ''I_WANT'', 'NEW_YORK'', or ''COULDN'T''. Index positions start at 1. A single number I can be given, implying ''(I I+1)''. The :frame argument identifes the frames of acoustic data covered by the word.

(tell :content (backto :index I))
Broadcast to indicate that any words previously output at index I or beyond (inclusive) are no longer valid parts of the hypothesis.

(tell :content (stop :uttnum N))
Broadcast to announce the end of the utterance.

ENVIRONMENT

TRAINS_BASE	Used to find Sphinx-II model files
AUDIOFILE	HOST:DEVICE for AudioFile server

FILES

$TRAINS_BASE/SpeechData	Location of default Sphinx-II files
sphinx.log	Stuff printed by sphinx if anyone cares
utt.NNN.au	Audio for utterance NNN
utt.NNN.out	Summary of recognition for utterance NNN

DIAGNOSTICS

Sometimes.

SEE ALSO

trains(1), tim(1), tspeechx(1), tspeechpp(1)

NOTES

Tspeech is actually a shell script that invokes the underlying Sphinx-II-derived program, tsphinx, with many of its parameters set appropriately. More subtle control can be had by changing this script, if you know what you're doing. See the manpage for fbs8_live(1) in /s7/sphinx-ii/man.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tspeechpp - TRAINS Speech Post-Processor

SYNOPSIS

tspeechpp [-s] [-v] [-t] [-nt] [-i] [-m] [-d] [-f2] [-fh] [-fi]
[-w file] [-2,3 file] [-c file] [-x file] [-in file] [-sc file]
[-mc file] [-l log-dir] [-q N]

DESCRIPTION

Tspeechpp is an error-correcting Speech Post-Processor. Its sole purpose is to boost the word recognition accuracy rates for a continuous speech recognition (CSR) system. It requires training on the behavior of the target CSR system in order to make appropriate corrections on unseen data. It was designed and implemented at the University of Rochester Department of Computer Science specifically for the TRAINS project. Although it has only been trained for and used with the Sphinx-II speech recognition system from Carnegie Mellon University, it can be used with other continuous speech recognizers. It can run standalone and interactively (when invoked with the -i argument) for testing, or it can communicate with other components of the TRAINS system via the TRAINS Input Manager. If desired, when running interactively and invoked with the -s argument, tspeechpp provides diagnostic statistics. These statistics can be made more verbose via the -v argument.

Tspeechpp is a "faceless" application, i.e., it has no display component itself. It responds to KQML messages on stdin that start and stop post-processing, and it broadcasts the results of the incremental processing to stdout.

OPTIONS

-s Specify that tspeechpp should provide diagnostic statistics to stdout. Off by default.

-v Specify that tspeechpp should provide verbose diagnostic output to stdout. Off by default.

-t Specify that tspeechpp should expect KQML messages on stdin and format its output as such messages on stdout. This is the default.

-nt Specify that tspeechpp should expect raw words on stdin and provide only raw words on stdout. This is off by default but can be used for interactive testing of the internal search process or for batch experiments. Off by default.

-i Specify that tspeechpp should provide a prompt to an interactive user. Off by default.

-m Specify that tspeechpp should use a minimum log probability threshold in its search among likeley corrections in order to prune the search space. On by default in the installed version; therefore this switch is actually redundant as installed.

-d Specify that tspeechpp should dump its internal state at the end of an utterance. This is a more recent addition than the "-n" or "-g" features and is probably more reliable.

-f2 Specify that tspeechpp should use a 1->2 fertility model component in its channel model.

-fh Specify that tspeechpp should use a 2->1 (half) fertility model component in its channel model.

-fi Specify that tspeechpp should use a 0->1 (insertion) fertility model component in its channel model.

-c file
Specify the confusion (1->1) channel model to be used. The default is the "TDC-75" model.

-w file
Specify the wbic (word bigram count) file to be used. The default is the "TDC-75" model.

-x file
Specify the deletion (1->0) file to be used. The default is the "TDC-75" model. This is currently unused.

-sc file
Specify the split-candidates (1->2) file to be used. The default is the "TDC-75" model.

-in file
Specify the insertion (0->1) file to be used. The default is the "TDC-75" model.

-mc file
Specify the merge-candidates (2->1) file to be used. The default is the "TDC-75" model.

-2|3 file
Specify the back-off language model file to be used, and specify whether bigrams (-2) alone or trigrams (-3) also should be used. The default is the "TDC-75" model.

-l dir
Specify the log dir, where the initial log file should be written.

-q N
: Specify the depth (as a small positive integer) of the priority queue for maintaining top alternate hypotheses. For valid "N-best" hypotheses, this number must be N.

USAGE

Tspeechpp first loads its channel and language model files.

Receipt of a START message initiates post-processing. Words are broadcast as they are corrected, with a BACKTO message being used to indicate revision of a word previously output. The last word of the hypothesis is buffered to prevent excessive BACKTO's on the frontier of the correction process. This process is complicated by the fact that BACKTO's also appear on the input as they come from tsphinx. Receipt of an END message terminates the utterance; tspeechpp finishes processing the utterance, and outputs any remaining words, followed by an END message.

TSPEECHPP MESSAGES

The following KQML messages are understood by tspeechpp. They should be addressed with ":receiver SPEECH-PP". Case is insignificant outside of strings. Whitespace between messages is ignored.

(request :content (start))
: Start correction. Initiates broadcast of corrected words.

(request :content (end))
: Completes processing of the current utterance.

(request :content (exit))
: Request that tspeechpp exit.

(request :content (offline t))
: Request that tspeechpp go offline and ignore all subsequent messages except an EXIT.

(request :content (offline nil))
: Request that tspeechpp come back online into full activity.

(request :content (chdir path-to-logdir))
: Request that tspeechpp change its log directory and file as specified by the given path.

The following messages are generated by tspeechpp:

(tell :receiver PM :content (ready))
: Sent to the PM to indicate the module is ready for action.

(tell :content (word STRING :index (I J) :uttnum N))
Broadcast to announce a new word in the hypothesis. The index identifies the start and end position of the word, which can in fact be several words as far as the parser is concerned, as in the token ''I_WANT'', 'NEW_YORK'', or ''COULDN'T''. Index positions start at 1. A single number I can be given, implying ''(I I+1)''.

(tell :content (backto :index I))
Broadcast to indicate that any words previously output at index I or beyond (inclusive) are no longer valid parts of the hypothesis.

ENVIRONMENT

TRAINS_BASE Root of TRAINS distribution

FILES

$TRAINS_BASE/SpeechPP/models Location of default Speech-PP files

DIAGNOSTICS

Sometimes.

SEE ALSO

trains(1), tim(1), tspeech(1)

NOTES

Tspeechpp is a Perl version 5 program. It was originally implemented in Perl for prototyping. A port to C was never attempted as Perl seemed to suffice, however, speed is now becoming an issue with the complex channel models.

Tspeechpp has been documented in several papers, including:

Eric K. Ringger and James F. Allen. "A Fertility Channel Model for Post-Correction of Continuous Speech Recognition." Proceedings of the Fourth International Conference on Spoken Language Processing (ICSLP'96). October, 1996.

AUTHOR

Eric Ringger (ringger@cs.rochester.edu).

NAME

tspeechx - TRAINS Speech Controller

SYNOPSIS

tspeechx [-rows N] [-columns N] [-stopdelay N] [-clickAndHold BOOL] [-showMenus BOOL] [-showLabels BOOL] [-showSpeechIn BOOL] [-showSpeechPP BOOL] [-fontpat STR] [-fontsize N] [-debug where] [X args]

DESCRIPTION

Tspeechx is the TRAINS Speech Controller. It provides an X/Motif display that includes windows for viewing the results of speech recognition (both SPEECH-IN and SPEECH-PP) and a button for starting and stopping the recognition (which it does by sending messages to SPEECH-IN).

OPTIONS

-rows N
: Number of rows for each of the result windows (default 4).

-columns N
: Number of columns for each of the result windows (default 25).

-stopdelay N
: Number of microseconds between when the user releases the "Talk" button and when the STOP message is sent. The default is 500000 (half a second). This is useful to prevent people cutting off the end of their utterances.

-clickAndHold BOOL
: If True (the default), a START message is sent to SPEECH-IN when the "Talk" button is pressed and a STOP message is sent when it is released. If False, one click sends the START message and a second click sends the STOP message. The label on the "Talk" button is set accordingly.

-showMenus BOOL
: Enable (True) or disable (False) the display of the application menubar.

-showLabels BOOL
: Enable (True) or disable (False) the display of the module names and their status.

-showSpeechIn BOOL
: Enable (True) or disable (False) the display of SPEECH-IN results. Disabling the display makes the Speech Controller window smaller.

-showSpeechPP BOOL
: Enable (True) or disable (False) the display of SPEECH-PP results. Disabling the display makes the Speech Controller window smaller.

-fontpat STR
: Set the font pattern to the given STR, which should have a single "default is a pattern representing Helvetica bold.

-fontsize N
: Set the initial font size to N. The default is 18.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args
: Tspeechx accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Tspeechx sends messages to the Input Manager to LISTEN to the output of SPEECH-IN and SPEECH-PP, as well as MONITORing their status. The output words are processed and displayed in text windows; the status is displayed using a colored button. Either display can be hidden using the menus, and in addition the Speech Post-Processor can be taken offline, causing it to ignore messages until it is put back online.

The tspeechx "Talk" button can be operated in either of two modes, depending on the -clickAndHold option value. If this is True (the default), a START message is sent to SPEECH-IN when the "Talk" button is pressed and a STOP message is sent when it is released. If it is False, one click sends the START message and a second click sends the STOP message. The label on the "Talk" button is set accordingly. See also tkeyboard(1) for a means of starting and stopping speech recognition without using the mouse.

SPEECH CONTROLLER MESSAGES

The following KQML messages are understood by the Speech Controller. They should be addressed with ":receiver SPEECH-X". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(tell :content (start :uttnum N))
: Sent by speech recognition modules at the start of an utterance. Clears the appropriate window and sets the utterance counter.

(tell :content (word W :uttnum N :index I :frame F :score S))
Adds a word to the display for the appropriate module.

(tell :content (backto :uttnum N :index I))
Invalidates words at index positions I and higher (inclusive) for the appropriate module.

(tell :content (input-end :uttnum N))
Sent by SPEECH-IN when it receives the STOP message. Ignored.

(tell :content (end :uttnum N))
Sent by speech recognition modules at the end of an utterance.

(reply :content (status M S))
Sent by the Input Manager when the status of SPEECH-OUT or SPEECH-PP changes. Used to change the color of the status indicator on the Speech Controller panel.

(request :content (reset))
Clears the Speech Controller display and resets any internal state.

(request :content (exit :status N))
Requests that the Keyboard Manager exit with status N (default 0).

(request :content (hide-window))
Causes the Speech Controller display to iconify itself.

(request :content (show-window))
Causes the Speech Controller display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Treated as a RESET followed by SHOW-WINDOW.

(request :content (end-conversation)
Treated as a HIDE-WINDOW.

(request :content (set-button)
Causes the "Talk" button to be highlighted. Useful for replay.

(request :content (unset-button)
Causes the "Talk" button to not be highlighted. Useful for replay.

(request :content (chdir DIR))
Ignored.

The following messages are generated by tspeechx to control speech recognition.

(request :receiver SPEECH-IN :content (start))
Sent when the user first presses the "Talk" button.

(request :receiver SPEECH-IN :content (stop))
Sent when the user releases the "Talk" button (in click-and-hold mode), or on the second click (in click-to-talk mode).

ENVIRONMENT

DISPLAY HOST:SCREEN for X server

FILES

None.

DIAGNOSTICS

Nope.

SEE ALSO

trains(1), tspeech(1), tspeechpp(1)

BUGS

I don't actually think so.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tsplash - TRAINS Splash Screen module

SYNOPSIS

tsplash [-pixmap STR] [-logdir STR] [-name STR] [-lang STR] [-sex m|f] [-intro BOOL] [-scoring BOOL] [-asksave BOOL] [-debug where] [X args]

DESCRIPTION

Tsplash is the TRAINS Splash Screen module. It provides an X/Motif window displaying the "splash screen"--a pretty picture with the system logo and credits, suitable for leaving displayed on an idle machine. It allows the user to specify their name, language, sex, and session options, and then start a session by pressing the "Start" button. At the end of a session, the user is allowed to save some or all data generated by the session as well as sending email to the maintainers. A "Help" button plays an introductory QuickTime tutorial and a "Quit" button allows the system to be shut down.

Note: The male/female toggle buttons on the splash screen do NOT currently affect the models used by the speech recognizer, tspeech(1). You need to set the enviornment variable TRAINS_USER_SEX, as documented in the tspeech manpage, prior to starting TRAINS. To change the models online, you need to kill and restart tspeech, perhaps by using the tshortcuts(1) module.

OPTIONS

-pixmap STR
: Specify the name of the pixmap (XPM) file to display in the splash screen. Changing the size from the default may or may not work properly...

-logdir STR
: Specify the directory into which logs from TRAINS sessions should be stored. The default is the value of the environment variable TRAINS_LOGS, or TRAINS_BASE/logs if the former is not defined.

-name STR
: Specify the initial user name. The default is the user's login name as extracted from the password file.

-lang STR
: Specify the initial user language. The default is "US English".

-sex m|f
: Specify the initial setting of the male/female toggle buttons. The default is "m". See the note above regarding the irrelevance of this setting.

-intro BOOL
: Specify the initial value of the "Intro" checkbox. When True, the system may play a tutorial introduction for new users. The default is False.

-scoring BOOL
: Specify the initial value of the "Scoring" checkbox. When True (the default), the system will perform the scoring phase at the end of a conversation.

-asksave BOOL
: If True (the default), the user will be prompted at the end of a conversation to save some or all of their session data and/or send email. If False, all data is saved unconditionally. This is good for demos.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args
: Tsplash accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Tsplash starts by loading its splash screen pixmap and displaying it with the user information panel. When the user hits Return or clicks on the "Start" button, tsplash creates a new log directory and writes the file "user" in it containing the user information entered on the panel. It then broadcasts a CHDIR request to all modules in the system via the Input Manager (using a broadcast performative), and then broadcasts a START-CONVERSATION message to initiate the conversation. Finally, it hides itself by iconifying its toplevel window.

Upon receipt of an END-CONVERSATION message, tsplash redisplays its window and, unless -asksave was False, displays the panel that allows the user to save some or all of their session and/or send email. Once the user dismisses this panel, tsplash waits for the user to either start a new conversation or quit.

Clicking the "Quit" button causes tsplash to send an EXIT request to the Input Manager, effectively halting the session. Clicking the "Help" button causes it to run the program TRAINS_BASE/bin/tintro. If configured properly, this should play a tutorial QuickTime movie.

SPLASH SCREEN MESSAGES

The following KQML messages are understood by the Keyboard Manager. They should be addressed with ":receiver SPLASH". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (exit :status N))
: Requests that the Splash Screen exit with status N (default 0).

(request :content (hide-window))
: Causes the Splash Screen display to iconify itself.

(request :content (show-window))
: Causes the Splash Screen display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
: Causes the Splash Screen to act as if the user had pressed the "Start" button (i.e., creates a new log directory, broadcasts CHDIR and START-CONVERSATION, and hides itself).

(request :content (end-conversation)
: Causes the Splash Screen to redisplay itself and, unless -asksave was False, display the Save/Email panel.

(request :content (chdir DIR))
: Ignored.

ENVIRONMENT

DISPLAY	HOST:SCREEN for X server
TRAINS_LOGS	Directory for log files

FILES

user	User information logfile

DIAGNOSTICS

May complain about colormap allocations.

SEE ALSO

trains(1), tim(1)

BUGS

Not sure we got the intro movie thing working properly...

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

ttc - TRAINS TrueTalk client using AudioFile server

SYNOPSIS

ttc [-h host] [-audio server] [-debug where]

DESCRIPTION

Ttc is a TrueTalk (tm) client for use with the TRAINS system. That is, it connects to a TrueTalk server to provide speech generation services for TRAINS. This version of ttc uses the AudioFile server for network audio facilities.

OPTIONS

-h host
: Connect to the TrueTalk server running on host. The default is find it on the current host. The server must be running and ready to accept connections before ttc starts (see USAGE, below).

-audio server
: Connect to the named AudioFile server. The default is the value of the environment variable AUDIOFILE, if set, otherwise the local machine.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

USAGE

The TrueTalk server must be running before ttc starts. Within TRAINS, this is usually accomplished by the tttalk(1) script. However, if you wanted to do it yourself, you could run the following commands:

```
% setenv TT_BASE /s5/truetalk/tts.r1.`uname -s`.`uname -r`
% $TT_BASE/bin/tts -server -i
```

That is, you need to set the environment variable TT_BASE, then run tts in server mode (-i sets incremental mode). The example above is based on the TrueTalk installation at the time this manpage was written.

For this version of ttc, you must also be running the AudioFile server. Again, this will normally be taken care of when ttc is used in the TRAINS system, however to run it by hand you can do:

```
% /u/trains/AF/bin/Asparc10 -rate 16000 &
```

You might of course need to run a different AudioFile server, depending on the platform. For example, Asparc (without the "10") would be used for Suns that don't have speakerbox audio.

With both servers running, you can then run ttc, specifying the server host using "-h" if necessary (if you ran the TrueTalk server on another machine). Ttc listens for KQML messages on its standard input, as described below, and outputs KQML error and reply messages to its standard output.

TTC MESSAGES

The following KQML messages are understood by ttc. They should be addressed with ":receiver SPEECH-OUT". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (say STRING))
: Sender is requesting that STRING be passed to TrueTalk (i.e., "spoken"). When the speech has been generated and the audio has finished playing, ttc will reply with a message of the form:

```
(reply :sender SPEECH-OUT :content (done))
```

Any :reply-with in the original message will be used in the reply.

(request :content (exit :status N))
: Requests that ttc exit with status N (default 0). This usually causes the TrueTalk server to exit also.

(request :content (chdir DIR))
: Ignored.

(request :content (show-window))
: Ignored.

(request :content (hide-window))
: Ignored.

(request :content (start-conversation :name N :lang L :sex S))
: Ignored.

(request :content (end-conversation)
: Ignored.

ENVIRONMENT

AUDIOFILE HOST:DEVICE for AudioFile server

FILES

None.

DIAGNOSTICS

Perhaps.

SEE ALSO

trains(1), tttalk(1), tts(1)

BUGS

It seems that the AudioFile sometimes needs to re-sync it's clocks, and I suspect that this may cause the client playback to get lost. It shouldn't last more than one utterance, but I think I may have seen it get more confused.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

ttcl - TRAINS Discourse Manager module

SYNOPSIS

```
ttcl [lisp args] -- [-socket HOST:PORT] [-debug t|nil]
    [-debug-interactive t|nil] [-score-p t|nil] [-seed N]
```

DESCRIPTION

Ttcl is a dumped Allegro Common Lisp image that implements the TRAINS Discourse Manager module. The DM module is responsible for taking the output of the parser (an utterance interpretation), determining what was really said, how it fits into the plan(s) under consideration, and what to do in response. It uses the Problem Solving (PS) module to perform the underlying reasoning, and generates requests to the DISPLAY and SPEECH-OUT modules for its responses.

It should be noted that the Discourse Manager in fact consists of several somewhat separate sub-modules that communicate using internal KQML messages.

Unlike the other modules in the TRAINS System, the Discourse Manager does not use standard input and standard output for KQML messages, preferring to save those streams for debugging. Instead, it establishes its own connection to the IM at initialization.

OPTIONS

-socket HOST:PORT
: Connect to Input Manager at the given location. The default is the value of the environment variable TRAINS_SOCKET.

-debug t|nil
: Enable or disable incredibly verbose tracing to the file "Dm.log". In fact, there are some intermediate, undocumented values for this option.

-debug-interactive t|nil
: Not sure what this is for.

-score-p t|nil
: Enable or disable the "scoring" phase at the end of conversations.

-seed N
: Initialize the seed of the random number generator which, in principle, should cause sessions to be relatively repeatable.

lisp args

You can also pass any arguments suitable for dumped Lisp images, before the double-hypen that indicates the start of user-defined options. I don't know why you would want to, however.

USAGE

I certainly can't begin to describe the operation of the Discourse Manager here. Some more details are in various technical reports.

DISCOURSE MANAGER MESSAGES

In this section I am only going to document the configuration messages understood by the DM between conversations (and a few control messages). It also understands the messages broadcast by the Parser, as well as the messages it exchanges with the Problem Solving module.

(request :content (start-conversation :name N :lang L :sex S))

Starts a conversation. Upon receiving this message, the Discourse Manager initializes the scenario, generates the initial greeting, and prepares to receive input.

(request :content (end-conversation))

Causes the DM to abandon the current conversation and wait for the next one to start.

(request :content (chdir DIR))

Request that the Discourse Manager close the current "Dm.log" and open a new one in the given DIR.

(request :content (exit :status N))

Requests that the Discourse Manager exit with status N (default 0).

(request :content (config :speech-in t|nil))

Informs the DM that speech input is or is not in use.

(request :content (config :speech-out t|nil))

Informs the DM that speech output is or is not in use.

(request :content (config :speech-rate N))

Sets the speaking rate to N (a floating point value, default 1.0). This is passed TrueTalk via SPEECH-OUT.

(request :content (config :personality P))

Selects a personality. Possible values for P are: :casual, :abusive, :humorous, :paranoid, :respectful, :dry, and :snide. I believe that emotions must be enabled for personalities to have any effect.

(request :content (config :emotions t|nil))

Enables or disables emotions.

(request :content (config :score t|nil))
Like the -score command line argument, this enables or disables the "scoring" phase at the end of conversations. The default is enabled.

(request :content (config :intro t|nil))
Enables or disables the demonstrative introduction given by the system to new users. The default is enabled, I think.

(request :content (config :known-goals t|nil))
Determines whether the system "knows" the goals of the scenario in advance. The default is nil.

(request :content (config :start (city1 city2 ...)))
Sets start locations for engines in scenario. The default is 2 randomly-located engines.

(request :content (config :enum N))
Specifies N randomly-located engines in the scenario. The default is 2.

(request :content (config :goal (city1 city2 ...)))
Specifies destinations for scenario. The default is random destinations.

(request :content (config :seed N))
Like the -seed command line option, this initializes the random number generator.

(request :content (config :debug t|nil))
Like the -debug command line option, this detremines the level of logging to the "Dm.log" file.

(request :content (config :xcitydelay N))
Delay per train if routes cross at a city.

(request :content (config :xtrackdelay N))
Delay per train if routes cross share a track.

(request :content (config :ccities (city1 city2 ...)))
Specify list of congested cities. Can also be a list of pairs of the form "(CITY REASON)".

(request :content (config :ccnum N))
Specify that there should be N randomly-selected congested cities (with random reasons).

(request :content (config :ctracks (city1 city2 ...)))
Specify list of congested tracks. Can also be a list of pairs of the form "(TRACK REASON)".

(request :content (config :ctnum N))
Specify that there should be N randomly-selected congested tracks (with random reasons).

ENVIRONMENT

The dumped Allegro CL image depends on a shared library for execution. You can specify the location of this library using the environment variable ALLEGRO_CL_HOME if it is not in the same place as when the image was dumped.

FILES

Dm.log Discourse Manager log

DIAGNOSTICS

Copious logging. In the event of a crash, tends to print Lisp debugger information to stderr (or worse, stdout). You can occasionally get into the Lisp debugger, if -debug was not nil.

SEE ALSO

trains(1)

BUGS

Nah... ;-)

AUTHOR

Brad Miller (miller@cs.rochester.edu).

NAME

ttranscript - TRAINS Transcript module

SYNOPSIS

ttranscript [-nolog] [-debug where] [X args]

DESCRIPTION

Ttranscript is the TRAINS Transcript module. It provides an X/Motif window with an ongoing transcript of the current session. It also writes the file "transcript" in the log directory containing the same transcript.

OPTIONS

-nolog
: If given, prevents ttranscript from writing the "transcript" file. This can be useful during replay.

-debug where
: Specify that copious debugging information should be written to where. If where is a hyphen ("-"), output is to stderr, if it starts with a pipe ("|") the rest of the argument is passed to popen(3), otherwise it is taken to be a filename to which to write. Be careful to escape pipes and spaces from the shell.

X args
: Ttranscript accepts all standard X Toolkit arguments; see X(1) for details.

USAGE

Ttranscript sends messages to the Input Manager to LISTEN to the USER-INPUT class of modules. This class includes the speech recognition modules SPEECH-IN and SPEECH-PP, the Keyboard Manager KEYBOARD, and the DISPLAY. As these modules broadcast the user's spoken, typed, or graphical input, the Transcript module puts together a transcript of the session. The system's utterances are recorded in the transcript by the Discourse Manager using an explicit LOG request.

Note that having the transcript module monitor the speech recognizers and put together "what the user said" is not really the right thing to do (although it is the least intrusive). This is because in order to do it right, the Transcript module has to know and duplicate the approach taken by the Parser regarding the multiple input streams. A better approach would be to have the parser decide "what was said", and then log it to the Transcript.

TRANSCRIPT MESSAGES

The following KQML messages are understood by the Keyboard Manager. They should be addressed with ":receiver TRANSCRIPT". Case is insignificant outside of strings. Whitespace between messages is ignored (but rigorously enforced within messages according to the KQML spec).

(request :content (log STR))
: Adds the given string to the transcript (file and display). By convention, STR is of the form

```
WHO HOW text
```

where WHO is either SYS or USR, and HOW indicates how the utterance was conveyed (e.g., "dsp" for display, "txt" for typed input, etc.).

(tell :content (start :uttnum N))
: Sent by speech recognition modules at the start of an utterance.

(tell :content (input-end :uttnum N))
: Ignored.

(tell :content (word W :uttnum N :index I :frame F :score S))
: Adds a word to the current utterance. The format is more fully described in the manpages for tspeech(1), tspeechpp(1), and tkeyboard(1).

(tell :content (backto :uttnum N :index I))
: Invalidates words at index positions I and higher (inclusive).

(tell :content (end :uttnum N))
: Sent by speech recognition modules at the end of an utterance.

(tell :content (mouse :select obj1 obj2 ...))
: Sent by DISPLAY when the user clicks on an object.

(tell :content (mouse :drag obj :from obj :to obj1 obj2 ...))
: Sent by DISPLAY when the user drags an object.

(tell :content (confirm TAG t|nil))
: Sent by DISPLAY when the user answers a dialog box confirmer.

(request :content (exit :status N))
: Requests that the Transcript exit with status N (default 0).

(request :content (hide-window))
: Causes the Transcript display to iconify itself.

(request :content (show-window))
: Causes the Transcript display to deiconify itself.

(request :content (start-conversation :name N :lang L :sex S))
Like a SHOW-WINDOW but also rease the Transcript window.

(request :content (end-conversation)
Treated like HIDE-WINDOW.

(request :content (chdir DIR))
Causes the Transcript module to close its "transcript" file and open a new one in the given DIR.

ENVIRONMENT

DISPLAY HOST:SCREEN for X server

FILES

transcript Transcript file

DIAGNOSTICS

Too simple for diagnostics.

SEE ALSO

trains(1)

BUGS

I sure hope not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tts - Runs TrueTalk server

SYNOPSIS

tts [args]

DESCRIPTION

This script simply configures the environment as needed to run the TrueTalk (tm) server, then invokes it with the -server argument (and any other arguments given on the command line).

For details of the SPEECH-OUT module, see the manpage for ttc(1).

ENVIRONMENT

Appropriate defaults for the following variables are used if they are not set when tts is run. Their values may need to change if the TrueTalk configuration changes, for example in a travel system.

ELM_HOST	Host running license daemon
TT_BASE	Root of TrueTalk directory tree

FILES

None.

DIAGNOSTICS

Some messages when the server exits.

SEE ALSO

trains(1), tttalk(1), ttc(1)

BUGS

This is commercial software, not that that's a bug, but I thought I'd mention it. It is only licensed for certain machines, which can be difficult during road shows.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

tttalk - TRAINS Speech Generation module

SYNOPSIS

tttalk [-h host] [ttc args]

DESCRIPTION

Tttalk is a script that launches the TrueTalk server, tts(1), and client, ttc(1). These two processes between them provide speech generation services for the TRAINS System. Further details are available in their manpages.

OPTIONS

-h host
: Specifies the host on which to start the TrueTalk server tts(1). This argument is also passed to ttc(1) so it can find the server.

ttc args
: Any other arguments are passed to ttc(1).

USAGE

Running tttalk starts the TrueTalk server and client with standard input and output are connected to the client. Tttalk then waits for either process to exit, whereupon it kills the other and exits itself.

SPEECH-OUT MESSAGES

Messages understood by the SPEECH-OUT module are described in the manpage for ttc(1).

ENVIRONMENT

None (but see ttc(1) and tts(1)).

FILES

None.

DIAGNOSTICS

Not of its own.

SEE ALSO

trains(1), ttc(1), tts(1)

BUGS

Not of its own.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

libKQML - TRAINS System KQML Library

SYNOPSIS

```
#include <KQML.h>
```

DESCRIPTION

KQML is the Knowledge Query and Manipulation Language, a communication protocol for knowledge-based systems. The TRAINS KQML Library provides C data structures and routines for manipulating the KQML performatives exchanged between modules of the TRAINS System.

Basically, KQML performatives are parenthesized lists consisting of a "verb" (e.g., "tell" or "ask") and a sequence of keyword-value "parameters" (e.g., :sender, :content). KQML performatives are transmitted as ASCII strings, and the libKQML routines are essentially routines for parsing and managing these strings. For more details on KQML, see the specification and/or the TRAINS-96 Technical Note.

KQML SYNTAX

The following is a brief BNF grammar for KQML:

```
<performative> ::= ( <word> {<white> :<word> <white> <expr>}* )
<expr> ::= <word> | <quotation> | <string> |
           (<word> {<white> <expr>}*)
<word> ::= <char><char>*
<char> ::= <alphabetic> | <numeric> | <special>
<special> ::= < | > | = | + | - | * | / | & | ^ | ~ | _ |
              @ | $ | % | : | . | ! | ?
<quotation> ::= '<expr> | `<comma-expr>
<comma-expr> ::= <word> | <quotation> | <string> | ,<comma-expr> |
                 (<word> {<white> <comma-expr>}*)
<string> ::= "<stringchar>" | #<digit><digit>*"<ascii>*
<stringchar> ::= \<ascii> | <ascii-not-\-or-">
```

This grammar assumes definitions for <ascii>, <alphabetic>, <numeric>, etc. A "*" means zero or more occurrences, and braces indicate optional items. Note that <performative> is a specialization of <expr> (requiring strict keyword-value alternation).

This specification is quite restrictive. It prohibits the empty list "()", for example, as well as whitespace between the last element of a list and any of its closing parentheses. It also prohibits lists of strings (requiring a <word> at the start of any list). Some of these restrictions have been relaxed in TRAINS.

MEMORY FUNCTIONS

The following functions allocate and manipulate performatives.

KQMLPerformative *KQMLNewPerformative(char *verb)
: Allocates a new performative with the given verb and no parameters.

void KQMLFreePerformative(KQMLPerformative *perf)
: Frees the given performative and all its parameters.

char *KQMLGetParameter(KQMLPerformative *perf, char *key)
: Returns the value of the requested parameter, i.e., the element following KEY in the performative. Returns NULL is there is no KEY parameter. Note that this function does NOT allocate a copy of the string value.

KQMLParameter *KQMLSetParameter(KQMLPerformative *perf, char *key, char *value)
: Sets the value of the parameter KEY. This frees any previous value for the parameter and allocates a copy of the given VALUE.

PARSING FUNCTIONS

These routines convert between normal C strings (char*), string arrays (char **), and performatives.

char *KQMLParseString(char *in)
: Parses the input as a KQML string, either double-quoted or sharped, and returns a newly-allocated C string. Returns NULL if the input is not a KQML string.

char *KQMLParseQuotedString(char *in)
: Like KQMLParseString but the input must be a quoted KQML string.

char *KQMLParseSharpedString(char *in)
: Like KQMLParseString but the input must be a sharped KQML string.

char **KQMLParseList(char *in)
: Parses the input as KQML list and returns a newly-allocated, NULL-terminated array of C strings each of which is a newly-allocated copy of an element of the list. These elements are not themselves parsed, however. Returns NULL if the input is not a KQML list.

char **KQMLParseStringList(char *in)
: Like KQMLParseList, but further parses each of the elements of the list as KQML strings.

char *KQMLParseThing(char *in)
: Parses the input as a string if it is in KQML string syntax, otherwise simply returns a newly-allocated copy of the input (assumed to be a token). Returns NULL only if malloc(3) fails.

char **KQMLParseThingList(char *in)
Like KQMLParseList, but further parses each of the elements of the list that are KQML strings.

KQMLPerformative *KQMLParsePerformative(char *in)
Parses the input as a KQML performative and returns a newly-allocated KQMLPerformative structure containing the verb and any parameters. Note that a KQML performative is simply a list with an initial token followed by keyword-value pairs.

int KQMLParseKeywordList(char *in, char **keys, char ***vals)
For KEYS a NULL-terminated list of keywords, parses the input and places the newly-allocated copies of the corresponding values into corresponding elements of VALS. Returns the number of keywords matched.

INPUT FUNCTIONS

These functions are used for input of KQML performatives from open file descriptors. The enumerated type KQMLError is used to indicate errors encountered during processing.

KQMLPerformative *KQMLRead(int fd, KQMLError *errorp, char **txtp)
This routine reads a performative from the given file descriptor. It returns a newly-allocated performative and sets *ERRORP to 0 if successful. It returns NULL and sets *ERRORP to 0 if end-of-file is encountered. Otherwise it returns NULL and *ERRORP will be less than 0. If TEXTP is non-null, a newly-allocated copy of the text of the message (or text leading up to an error) is stored in *TXTP.

KQMLPerformative *KQMLReadNoHang(int fd, KQMLError *errorp, char **txtp)
This routine reads a performative from the given file descriptor without blocking in read(2). It returns a newly-allocated performative and sets *ERRORP to 0 if successful. It returns NULL and sets *ERRORP to a value greater than 0 if the performative is not yet complete. It returns NULL and sets *ERRORP to 0 if end-of-file is encountered. Otherwise it returns NULL and *ERRORP will be less than 0. If TEXTP is non-null, a newly-allocated copy of the text of the message (or text leading up to an error) is stored in *TXTP. Note that this is not only the text read this call--it is the text read thus far on this message, or up to this error.

char *KQMLErrorString(KQMLError num)
Returns a string corresponding to the given KQML error number. Note that this string is not freshly allocated.

MISCELLANEOUS FUNCTIONS

```
char *KQMLPerformativeToString(KQMLPerformative *perf)
```
Returns a newly-allocated string containing the text form of the given performative, suitable for printing.

```
KQMLPerformative *KQMLCopyPerformative(KQMLPerformative *perf)
```
Returns a newly-allocated copy of the given performative, with the same verb and parameters, all also copied.

SEE ALSO

trains(1), libtrlib(3), libutil(3)

BUGS

Swat. Ow! Swat. Yikes!

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

libtrlib - TRAINS System Module Library

SYNOPSIS

```
#include <debugarg.h>
#include <error.h>
#include <hostname.h>
#include <input.h>
#include <parse.h>
#include <send.h>
```

DESCRIPTION

The TRAINS System Module library, libtrlib, provides a collection of C language routines used in most, if not all, of the TRAINS modules. They generally provide functionality shared by all modules, such as waiting for KQML input, parsing it into messages, and generating responses.

DEBUG PROCESSING

The following function is used by all modules that understand the -debug command-line argument.

```
#include <debugarg.h>
```

```
void trlibDebugArg(int argc, char **argv)
```

Given a process' ARGC and ARGV, this function extracts the -debug option, if given, and sets up the FILE* debugfp used by the DEBUG* and ERROR* macros from libutil(3). If the argument is a hyphen ("-"), then debugging output is to stderr. If it starts with a pipe symbol ("|"), then the rest of the argument is passed to popen(3) and the resulting FILE* used for debugging output. Otherwise the argument is treated as a filename, which is opened for debugging output. Making this a library routine allows for uniform treatment of the option.

KQML ERROR PROCESSING

The following function is used to generate a KQML error performative in reply to a previous message.

```
#include <error.h>
```

```
void trlibErrorReply(KQMLPerformative *perf,
                     TrlibErrorCode code, char *comment)
```

Sends a KQML error performative with the given :code and :comment to the :sender of PERF. Any :reply-with in the original PERF will be used as the :in-reply-to of the reply.

HOSTNAME FUNCTIONS

It can be tricky to get your own hostname, so this function does it for you.

```
#include <hostname.h>
```

```
char *trlibGetHostname(void)
```

Returns the current hostname as a pointer to static storage.

KQML INPUT PROCESSING

All TRAINS modules need to process KQML messages, whether they block waiting for them or not (for example, X apps can't block as that would prevent the user from interacting with the display).

```
#include <input.h>
```

```
int trlibInput(int fd, int block, TrlibCallbackProc cbp)
```

Reads KQMl input from fd. The block parameter should be one of TRLIB_BLOCK or TRLIB_DONTBLOCK. Returns less than 0 on error, 0 on end-of-file, and greater than 0 otherwise. In addition, if a performative is successfully read and parsed, and if the callback parameter cbp is non-NULL, it is called with the KQMLPerformative* as argument.

KQML PERFORMATIVE PARSING

All TRAINS modules need to look at the KQML performatives they receive and decide what to do, processing the ones they can and generating errors for those they can't. This function simplifies the first stages of that process.

```
#include <parse.h>
```

```
void trlibParsePerformative(KQMLPerformative *perf,
                            TrlibParseDef *defs)
```

The parameter DEFS describes the messages that the module expects to receive using a array of the following structures:

```
typedef void (*TrlibParseCallbackProc)(KQMLPerformative *perf,
                                       char **contents);
        typedef struct _TrlibParseDef {
            char *verb;
            char *content0;
            TrlibParseCallbackProc cb;
        } TrlibParseDef;
```

The last element in the array should have a NULL verb. The function then scans the array comparing first the verb of the performative then the first element of its :content (a NULL content0 in an entry means ignore the content). If a matching entry is found, its callback is called (a NULL cb in the entry means ignore the message completely). The callback can then further parse the contents of the performative. If no entry matches, an error is generated using trlibErrorReply(3).

KQML OUTPUT FUNCTIONS

All TRAINS modules need to send KQML performatives. The following function simplies this.

```
#include <send.h>
```

```
void trlibSendPerformative(FILE *fp, KQMLPerformative *perf)
```

Formats PERF to the given standard i/o file pointer. Note that this may block, although it's pretty unlikely.

SEE ALSO

trains(1), libutil(3), libKQML(3)

BUGS

Could well be...

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).

NAME

libutil - TRAINS System Utility Library

SYNOPSIS

```
#include <bitops.h>
#include <buffer.h>
#include <debug.h>
#include <error.h>
#include <memory.h>
#include <nonblockio.h>
#include <streq.h>
```

DESCRIPTION

The TRAINS utility library, libutil, provides a collection of C language routines I have found useful during development of many of the modules.

BIT OPERATIONS

The following macors are useful for manipulating bitsets.

```
#include <bitops.h>
```

BITSET(B, N)
: This macros sets the Nth bit in B.

BITCLR(B, N)
: This macros clears the Nth bit in B.

BITSET(B, N)
: This macros tests the Nth bit in B (that is, is non-zero if the bit is non-zero).

BUFFER OPERATIONS

The following functions operate on a opaque datatype Buffer that implements a dynamically-extensible circular character buffer.

```
#include <buffer.h>
```

Buffer *bufferCreate(void)
: Allocates and returns a new empty Buffer.

void bufferDestroy(Buffer *this)
: Frees up a previously allocated Buffer.

char *bufferData(Buffer *this)
: Returns a pointer to the data stored in the Buffer. Note that the data is NOT copied.

int bufferDatalen(Buffer *this)
Returns the number of bytes stored in the Buffer.

int bufferAvail(Buffer *this)
Returns the available space in the Buffer.

int bufferEmpty(Buffer *this)
Returns non-zero if the Buffer is empty.

int bufferAdd(Buffer *this, char *s, int len)
Adds LEN bytes from S to the Buffer.

int bufferAddString(Buffer *this, char *s)
Adds the contents of the NUL-terminated string S to the Buffer.

int bufferAddChar(Buffer *this, char c)
Adds a single character C to the Buffer.

int bufferDiscard(Buffer *this, int len)
Discards the first len characters from the Buffer.

int bufferGet(Buffer *this, char *s, int len)
Retrieves LEN characters from the buffer, stores them in S, and discard them from the Buffer.

void bufferErase(Buffer *this)
Makes the Buffer empty.

int bufferIncRefCount(Buffer *this)
Increments the Buffer's reference count.

int bufferDecRefCount(Buffer *this)
Decrements the Buffer's reference count.

int bufferRefCount(Buffer *this)
Returns the Buffer's reference count.

DEBUGGING MACROS

The following macros are useful for debugging. They must be compiled with the symbol DEBUG defined. Then, during execution, if the stream debugfp is non-NULL, they print debugging messages to it using fprintf(3). Use of these macros requires the definition of the FILE* debugfp (which can be stderr), and of the char* program, which should be the argv[0] of the process.

```
#include <debug.h>
```

DEBUG0(S)
Debugging message with no arguments.

DEBUG1(S, A1)
Debugging message with one argument.

DEBUG2(S, A1, A2)
Debugging message with two arguments.

DEBUG3(S, A1, A2, A3)
Debugging message with three arguments.

DEBUG4(S, A1, A2, A3, A4)
Debugging message with four arguments.

ERROR MACROS

The following macros are useful for printing error messages. They must be compiled with the symbol DEBUG defined. They print error messages to stderr using fprintf and, if DEBUG was define at compile-time, they use the corresponding DEBUG macro to log the error as well. Use of these macros requires the definition of the char* program, which should be the argv[0] of the process.

```
#include <error.h>
```

ERROR0(S)
Error message with no arguments.

ERROR1(S, A1)
Error message with one argument.

ERROR2(S, A1, A2)
Error message with two arguments.

ERROR3(S, A1, A2, A3)
Error message with three arguments.

SYSERR0(S)
System error message with no arguments. Includes output of strerror(3).

SYSERR1(S, A1)
System error message with one argument.

SYSERR2(S, A1, A2)
System error message with two arguments.

MEMORY OPERATIONS

These functions and macros are usful wrappers for the C library memory allocation routines.

```
#include <memory.h>
```

gfree(P)
Free P if it is non-NULL by calling free(3).

char *gnewstr(char *s)
Allocates and returns a new copy of NUL-terminated string S.

void gfreeall(char **strs)
Frees all non-NULL elements of the NULL-terminated array strs, then frees strs itself.

char **gcopyall(char **strs)
Allocates and returns a copy of the NULL-terminated array of strings.

NONBLOCKING I/O MACROS

These macros provide a portable way to use non-blocking i/o.

```
#include <nonblockio.h>
```

MAKE_NONBLOCKING(FD)
Marks FD for POSIX-style nonblocking i/o. Under SunOS and Solaris, this means setting the O_NONBLOCK flag using fcntl(2).

ISWOULDBLOCK(E)
Tests if the given value (typically errno) is the "would block" error code. Under SunOS and Solaris, this means tesing for EAGAIN.

MAKE_BLOCKING(FD)
Very untested attempt at restoring blocking i/o status.

STRING COMPARISON MACRO

This macro is used by all components of the TRAINS system when doing string comparisons.

```
#include <streq.h>
```

STREQ(S1, S2)
When this is defined to be strcasecmp(3), string comparisons are case-insensitive.

SEE ALSO

trains(1), libtrlib(3), libKQML(3)

BUGS

Probably not.

AUTHOR

George Ferguson (ferguson@cs.rochester.edu).