SLS-Lite: Enabling Spoken Language Systems Design for Non-Experts

by

Jef Pearlman

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

Masters of Engineering in
Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2000

Author . . . .
Department of
Electrical Engineering and Computer Science
August 23, 2000

Certified by . . . .
James R. Glass
Principal Research Scientist
Thesis Supervisor

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

SLS-Lite: Enabling Spoken Language Systems Design for Non-Experts

by

Jef Pearlman

Submitted to the Department of
Electrical Engineering and Computer Science
on August 23, 2000, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in
Computer Science and Engineering

Abstract

In this thesis, I designed and implemented SLS-Lite, a utility for allowing non-experts to build and run spoken language systems. This involved the creation of both a web interface for the developer and a set of programs to support the construction and execution of the required internal systems. We concentrated on simplifying the task of configuring the language understanding components of a spoken language system. Any application-specific functionality required could be provided by the developer in a simple, CGI-based back-end.

By learning the required grammar from a set of simple concepts and sentence examples provided by the developer, we were able to build a system where non-experts could build grammars and speech systems. Developers could also easily specify hierarchy in domains where a more complex grammar was appropriate.

We demonstrated SLS-Lite by building several domains ourselves, and allowing others to build their own. These included domains for controlling the appliances in a house, querying a directory of people in MIT's Laboratory for Computer Science, manipulating fictional objects, asking questions about music, and planning airline flights.

Thesis Supervisor: James R. Glass
Title: Principal Research Scientist

Acknowledgments

First and foremost, I would like to thank my thesis advisor, Jim Glass, for all of his help, direction, and above all, patience in guiding me through completion of my Masters thesis. Both his time and understanding have been invaluable to me.

I would also like to thank all of the staff and students in the Spoken Language Systems group for the help they have given me. Stephanie Seneff provided endless assistance with using and expanding TINA, and also provided an excellent introduction to the group by advising my Advanced Undergraduate Project. Chao Wang provided the tools and the knowledge to use the Hierarchical N-Gram language model. Matthew Mishrikey built the music domain, unconvering many issues with designing SLS-Lite along the way. Eugene Weinstein built the LCS-Info domain, uncovered even more issues with SLS-Lite, and will be continuing work on SLS-Lite in the future.

In addition, I would like to than Victor Zue and the rest of the Spoken Language Systems group for providing a productive and supportive research environment for me.

Finally, I would like to thank my mother, my father, and my brother, and well as my extended family and closest friends for their help and understanding in all of my work, my play, and my ceaseless wanderings throughout college and life. They are what make it all worthwhile.

This work was supported by DARPA under contract N66001-99-1-8904 monitored through Naval Command, Control, and Ocean Surveillance Center.

Contents

List of Figures

Chapter 1

Introduction

This thesis discusses SLS-Lite, a utility designed to allow non-speech-experts to build and use spoken language systems without having to learn all the complexities involved in creating and configuring system components. Over the course of this thesis, we will discuss what motivated us to build SLS-Lite, what decisions and problems we faced along the way, and how the system works in its current form.

1.1 Motivation

Spoken language systems are starting to become commonly used in applications ranging from automated answering services to stock trading to weather reporting [22] [2]. As speech technology improves, the number of applications where it can be effectively used increases. However, most speech technology still requires an expert or team of experts to utilize. Configuring such systems has traditionally required a lot of background knowledge and training. This limits the distribution of speech technology greatly – although many have potential applications for speech, only a few have the necessary knowledge to attempt to actually make as much use of the technologies as they would like.

Our goal with SLS-Lite was to develop a utility which would make it easier for developers without an extensive speech background to build, test, and use systems based on spoken interaction with computers. Making it easier for people to build and use these systems would get speech technologies into wider distribution, which would in turn have a tremendous number of benefits.

First of all, wider use of our technology can help us improve it. A variety of developers means a variety of domains. Since a lot of work goes into developing each individual domain in-house, allowing outside people to build their own domains gives us access to a much wider range of applications than we would normally have. This lets us see how a far greater variety of domains can be built, and learn what issues these new domains raise.

More people using our speech systems means more data collected. This mean a simple increase in the number of waveform samples we have, which in turn means better trained acoustic-phonetic models. Further, more users means more speakers from varied environments, giving us the data we need to build more robust systems in the future.

Allowing more people to use our systems from both a developer and a user standpoint also helps point out weaknesses in our current technologies, which helps determine which areas would benefit most from future research. For instance, through extensive public use of MIT's Jupiter [22] weather reporting system, we discovered that we needed to make use of confidence scoring [10] to improve the system's ability to handle uncertainty in the recognizer.

Finally, SLS-Lite need not only be used for "black box" creation of systems by non-experts. Not only do we hope that SLS-Lite itself will scale to more complex systems, but that as developers learn about speech systems and MIT's GALAXY architecture in particular, they can use SLS-Lite to seed more complex domains before adding the more difficult functionality by hand. This will give them the full power of the GALAXY system with the ease of creating a domain that is provided by SLS-Lite.

1.2 Goals

We had three major goals for SLS-Lite. The first was that it be easy to use for developers. Since our motivation for SLS-Lite was to enable non-experts to build speech systems, it had to be as easy to use as possible. This made it necessary for SLS-Lite to hide as much of the underlying complexity as it could. To this end, we chose to do things like use a web-based interface and example based learning.

Second, SLS-Lite needed to be flexible. The fewer ways in which we restricted the developer, the better it would be. Since our goal was to get as many people as possible using the system, any limitations in what the system could do would reduce the size of the audience we could reach, and thereby reduce the effectiveness of SLS-Lite. To make SLS-Lite as flexible as possible, we concentrated on making it work for conversational, mixed-initiative dialogues, rather than purely machine-directed interactions. We allowed the developers to make their grammars as complex as needed – from simple, flat grammars to complex, hierarchically-structured ones – all while maintaining a consistent, easy-to-use interface. We also allowed the construction of an arbitrary back-end, meaning that the developer could do as much or as little as they needed to deal with a user's request once it was received.

Finally, SLS-Lite needed to be robust. SLS-Lite needed to be as accurate as possible in its modeling of the user's interaction, while gracefully failing when it was unable to fully understand the user's input. To accomplish this, we used a robust statistical language model, known as a hierarchical n-gram, which closely matched the information the developer gave us. Further, when the system is unable to fully understand what the user said, it falls back to a "robust parse" [11], getting as much information as it can out of what the user spoke, and passing anything it learns to the back-end, letting the developer decide with how to respond most intelligently.

1.3 Solution

In order to accomplish our goals with SLS-Lite, we made several high level decisions. First, we decided to leverage off MIT's existing speech technologies. This gave us constantly-evolving, robust system, while in turn allowing things learned with SLS-Lite to further the developments of our own systems.

Also, in order to focus the project on a single task, we concentrated on building the language understanding technologies. This meant that we had to leave certain areas, such as discourse management, out of the initial version of SLS-Lite, although such areas may provide directions for future work.

Many other smaller choices were made in developing SLS-Lite, and these will be discussed throughout the rest of the thesis.

1.4 Evaluation

Since SLS-Lite is a tool for making it easier to build speech systems, there are no numeric metrics we can use to measure success. Instead we will judge the progress of SLS-Lite by the success of the people using it. As we built SLS-Lite, people within our research group were using it to build their own systems, and at the same time were exposing bugs and suggesting improvements for its development.

As SLS-Lite neared completion, we opened it up to other research groups within MIT. So far, we have had very positive feedback about the effectiveness of SLS-Lite. Ultimately, we hope to open SLS-Lite to the public, and improve it based on input from a wide variety of developers.

1.5 Examples

Throughout the description of SLS-Lite, we will be using three domains to illustrate various points. First, we will provide some backgrounds on the sample tasks. All three example domains are shown in detail in the appendices.

1.5.1 House Domain

The "house" domain is an attempt to show what a real non-hierarchical domain might be like. It covers the task of manipulating the appliances which might be found in a house. The goal is to be able to turn on and off as well as query the state of various appliances (such as lights) in several rooms – for instance, the user might say, "Please turn on the lights in the dining room," or ask, "Are the lights in the kitchen on?"

1.5.2 Toys Domain

The "toys" domain was used to test hierarchy within the SLS-Lite system. The system would accept commands from the user to move around a set a fictional objects. It should be able to deal with commands such as "Put the blue sphere on the table behind the red box," including ones (such as this example) which are ambiguous and can have several valid meanings an English.

1.5.3 LCS-Info Domain

The "LCS-Info" domain is an interface to a database of all of the people who work in MIT's Laboratory for Computer Science. The database should be able to understand and respond to queries such as, "What is Jef Pearlman's phone number?" In addition, it needs to be able to remember what has been said earlier in a conversation, so that if the user then asks, "What about his email address?" it knows whose email address it should retrieve.

Chapter 2

Background

The idea of creating development tools for building speech systems is not a new one. Several other groups have worked on similar projects, each with its own specific set of goals, strengths, and weaknesses.

2.1 VoiceXML

VoiceXML, the Voice eXtensible Markup Language, is a fairly new standard being endorsed by a forum of companies including AT&T, IBM, Lucent and Motorola [19]. Some companies, such as Tellme [16], are currently developing systems utilizing VoiceXML. Their on-line development system, Tellme Studio [17], is publicly available and allows experimentation with the technology.

> VoiceXML is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony, and mixed-initiative conversations. Its major goal is to bring the advantages of web-based development and content delivery to interactive voice response applications [20].

VoiceXML is based mainly on the concept of either "form" or "menu" dialogues. Forms provide a way to gather a set of inputs that the system needs, while menus provide a choice of what to do next. VoiceXML also provides a facility for designing a context free grammar (see Section 4.4 for more information on language models) for use by whatever speech recognizer is running the system.

VoiceXML is not, however, a program or a setup for running an entire spoken language system. It is a markup language which allows developers to describe dialogue interactions in a way that should let their description be used on any system that supports VoiceXML. Like SLS-Lite, much of the domain-specific logic can be provided in external CGI scripts. However, this product-independence means that the developer must deal with the configuration and execution of the necessary tools to actually run the system. This makes it similar to SLS-Lite in that it attempts to shield the developer from the implementation specifics, but different in that SLS-Lite is an attempt to actually provide a system to use, where VoiceXML requires the developer to find a third-party solution like Tellme Studio.

Since VoiceXML is simply a markup language which can be used by a variety of systems, it is possible that in the future SLS-Lite could actually use VoiceXML internally to describe some aspects of the interaction. Because SLS-Lite concentrates mostly on the interface to the developer, where VoiceXML describes an internal representation, the two are not incompatible – they can, in fact, serve complementary functions.

Building a speech system around VoiceXML still requires a basic understanding of building grammars – one of the major things SLS-Lite seeks to avoid. In order to use the context-free grammar facility of VoiceXML, the developer must understand how such a grammar is constructed – VoiceXML provides no guidance in the process. In SLS-Lite, the technical aspects of this process is hidden from the developer.

Also, unlike SLS-Lite, VoiceXML (at least in its current form) is mostly built for directed dialogues – support for mixed-initiative applications, while included, is minimal. While SLS-Lite does allow a directed dialogue type interaction, it was created more with the goal of conversational systems in mind, and is designed to make it easier to build that type of system.

2.2 CSLU Toolkit

CSLU, the Center for Spoken Language Understanding at the Oregon Graduate Institute of Science and Technology, has developed their own toolkit for designing and studying speech systems [4]. Rather than providing a language for developing dialogue systems, their toolkit gives developers a graphical user interface for building directed dialogue systems. By dragging around icons, entering sample queries and responses to build a context-free grammar, and using the provided software, developers can have a full speech system running on their desktop.

However, much like VoiceXML, the CSLU toolkit is aimed at designing directed dialogues, and requires the developer to run the full speech systems themselves. Although CSLU actually provides the software to do this, the developers must use their own hardware.

2.3 SpeechWorks and Nuance

Certain companies, such as SpeechWorks [15] and Nuance [9] have been developing their own proprietary tools. However, unlike the systems described above, and unlike SLS-Lite, they are currently not publicly available, although they may become so in the future. Their systems are mostly for use by customers or by their own internal developers. Some, like Nuance's commercial product V-Builder [18], are based on other technologies such as VoiceXML. Since V-Builder is a tool for developing VoiceXML, it has many of the same strengths and weaknesses as VoiceXML, as well as the same similarities and differences to SLS-Lite.

Like VoiceXML and the CSLU Toolkit, both companies' products are mostly aimed at developing directed dialog applications. Also, in both cases, the goals of the products differ greatly from SLS-Lite. The purpose of these internal development environments is to make the rapid development of speech systems to be deployed by the company or their customers, whereas the purpose of SLS-Lite is to allow almost anyone to build and use speech systems.

2.4 SLS-Lite

A major difference between a lot of previous work and our own is our concentration on simplifying the setup of the language understanding aspects of speech recognition. Many of these other tools assume a solid understanding of grammars and other parts of a speech system, while SLS-Lite is designed for someone with a very limited understanding of the underlying technologies. A developer does not need to understand how grammars are built in order to create one with SLS-Lite – they simply need to define a few concepts and give examples. They might not even be aware that they are describing a grammar in the process. In all of the other systems described, they must understand the process of building a grammar if they want to do so.

Also, many of these other tools are designed to make the construction of directed dialogue systems easier. However, we were not concerned with directed dialogue as much as with conversational systems. This concentration of mixed-initiative dialogue affected many of the design decisions made during the development of SLS-Lite.

Finally, our goal is to get as many outside groups as possible to use our system, both to our benefit and theirs. Some of the other tools which have been built are developed by companies only for in-house development. Although some systems (like Tellme Studio) do provide facilities to actually run domains, most of these other toolkits require outside systems capable of running full speech recognition and dialogue systems, or at least the knowledge to configure such a system. SLS-Lite, however, requires only a lightweight back-end to provide the application-specific functionality – the rest is provided by us.

Chapter 3

Architecture

The goal of SLS-Lite was to make it easier for developers to use MIT's human language technologies to make their own applications. MIT's systems are currently built on a system which was created and developed at MIT, and which is known as GALAXY [13]. The DARPA Communicator Program chose GALAXY as a common framework for participating researchers to use. In order to leverage MIT's current technology, we built SLS-Lite on the existing GALAXY infrastructure.

Since the GALAXY system is designed to be flexible, it should not limit what we can do with SLS-Lite, now or in the future. Because MIT has been using the GALAXY system in much larger and more complex systems than those which can currently be made with SLS-Lite, we know that it is up to the task of running such systems, and that it should scale with SLS-Lite to support these more complex domains. Finally, because GALAXY is in constant development by MIT and other DARPA Communicator contributors, basing SLS-Lite on GALAXY components will allow us to make use of improvements in the underlying technology without major modification of SLS-Lite itself.

3.1 Galaxy Architecture

Galaxy is a client-server architecture, where a central programmable "hub" server polls a set of human language technology (HLT) and other "spoke" servers, and decides where and when to send individual "frames" of data. Figure 3-1 depicts the general GALAXY architecture, along with several commonly used server components.

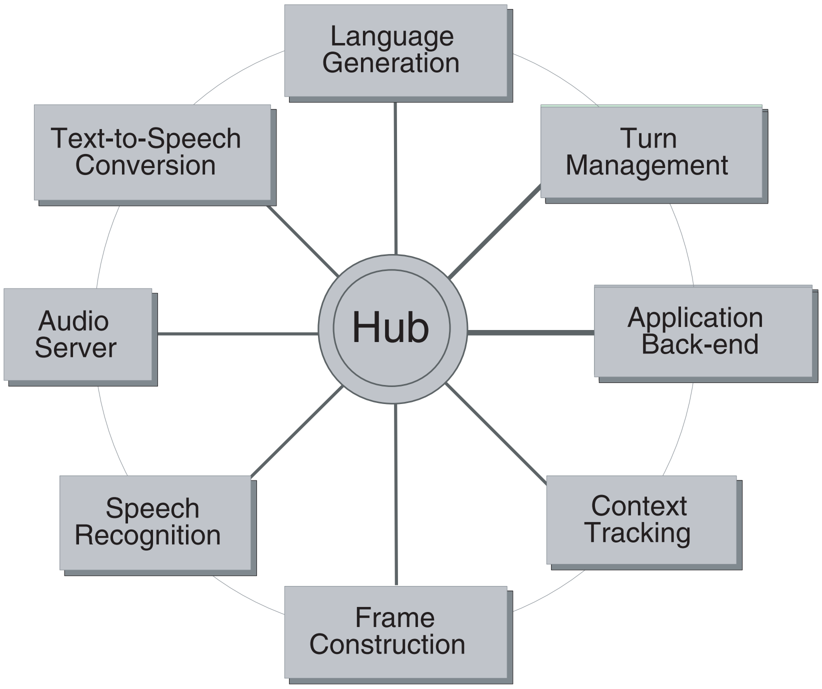

Figure 3-1: Architecture of GALAXY.

Below, we will describe some of the standard components of MIT's GALAXY system, and explain which of these we modified, automated configuration of, ignored, or replaced entirely in SLS-Lite.

3.1.1 Application Back-End

The application back-end is the part of the system which actually provides the domain-specific functionality. For instance, if the user asks for the weather in Boston within the JUPITER [22] system, the back-end actually looks up the data and formulates an appropriate response.

In the case of SLS-Lite, the application back-end is actually a GALAXY server which sits in between the developer-provided functionality and the hub. The actual application-specific work is done by a CGI script accessed through HTTP with a set of arguments which is defined by the developer's input. The script receives these arguments and provides a reply to be spoken to the user and, optionally, a history argument to be provided to the script on the next utterance.

3.1.2 Hub

The hub is a server governed by a script which controls all interaction between the various GALAXY components [13]. This central server makes network requests of the various subsystems. These subsystems process the requests and pass responses back to the hub, which decides what should be done with the data – whether it should be stored somewhere, passed on to another server, or ignored entirely.

For SLS-Lite, we made a hub script tailored specifically to the set of support servers we needed to use. It accepts incoming connections and/or phone calls, and manages the interaction between the audio server, speech recognizer, natural language generation and understanding servers, and speech synthesis components, as well as a new component which we used to connect to a remote CGI script, which in turn provided the domain-specific functionality.

3.1.3 Audio Server

The audio server connects to a networked audio client or telephone interface, sending it data received from a text-to-speech system or getting data to send to an automatic speech recognizer.

For most applications, MIT currently uses a telephone-based audio server which interfaces with Dialogic audio hardware. To talk to the system, the user must call a specific number and talk over the telephone in order to connect to the system. While this works well in many cases, other times it would be preferable to allow a user to connect to the system using their computer. To this end, MIT already has a Linux-based native audio server. For this project, we also build a Java-based recorder which could connect to the recognizer, and modified the Linux-based audio server to run on the Win32 platform. However, for SLS-Lite, the primary interface was telephone-based.

3.1.4 GUI Server

GALAXY can make use of a graphical user interface (GUI) server, which presents information to the user using text and graphics rather than speech. This can involve showing a web page, drawing a map, or coming up with some other representation of information relevant to the user.

SLS-Lite does not currently make use of a GUI server, since it has no generalized way to present data to the user other than through speech. However, in the future, SLS-Lite could provide the ability to the developer to build an external GUI which the user could then interact with.

3.1.5 Speech Recognition Server

The speech recognizer takes speech samples from the audio server and attempts to determine what words were spoken by the user. It generates a set of word and sentences hypotheses which it believes were most likely to have been spoken.

We used MIT's SUMMIT [6] speech recognizer to run SLS-Lite systems. Our particular setup of the recognizer uses a hierarchical n-gram grammar to describe how utterances might be formulated by the user. In addition to providing constraints on how people might speak, this grammar can be trained probabilistically to better model the likelihood of a particular set of words appearing in the hypothesized order. One of the major tasks of the SLS-Lite system performs is to build a grammar described by the developer, but in the format used by SUMMIT. We also train this grammar on the examples given by the developer. This process is explained in Section 4.3.

3.1.6 Natural Language Understanding Server

The natural language understanding and frame construction subsystem takes a sentence hypothesis (usually generated by the speech recognizer), parses it, and generates a "semantic frame" from the text. This is a form of meaning representation which attempts to encapsulate the conceptual information in the sentence. SLS-Lite uses

MIT's TINA [12] server for this purpose. A parse tree and the corresponding sample frame generated by TINA for SLS-Lite are shown in Figures 3-2 and 3-3.

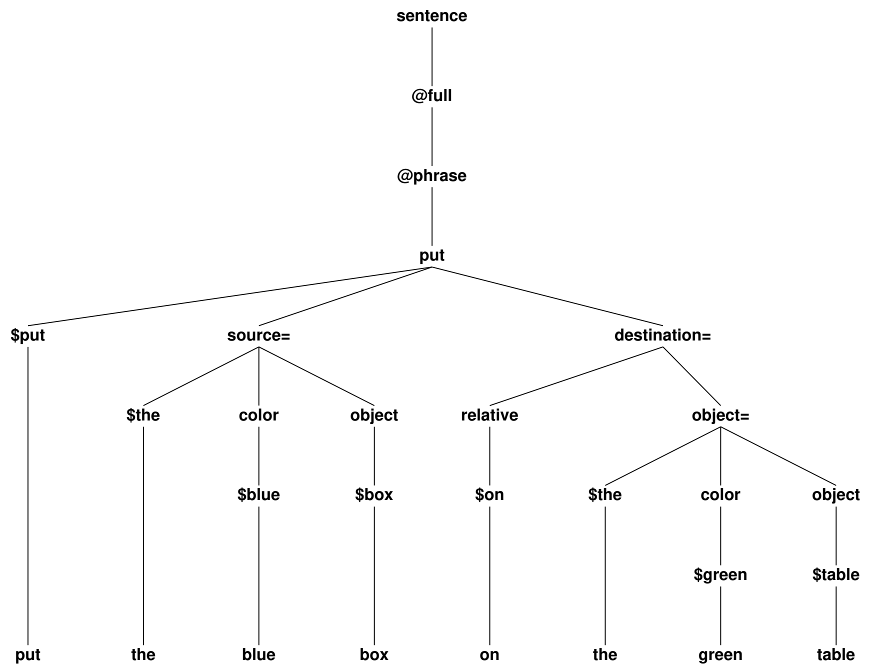

Figure 3-2: TINA parse tree for for sentence "Put the blue box on the green table."

The semantic frame divides the structure of the sentence up into a single "clause", and a set of "predicates" and "keys", which are used internally by both TINA and GENESIS (see Section 3.1.8 for more on GENESIS). In SLS-Lite, semantic frame keys correspond to important concepts within the sentence, where predicates correspond to hierarchical divisions. The single clause corresponds with the example given by the developer which matched the user's utterance.

It is important to note that Figure 3-3 is not the conventional semantic frame setup used by MIT. TINA also has a concept called a "topic", which aids in discourse management. A semantic frame utilizing MIT's standard setup might look like the one shown in Figure 3-4. Although SLS-Lite does not currently use topics or do any internal discourse regulation, future additions to the system might change this internal representation to use topics and perform some automatic discourse management.

To generate a frame from an utterance, TINA needs a set of rules files which describe the possible structures of utterances, including a description of what parts

```
{c put
   :pred {p source==
            :color "blue"
            :object "box" }
   :pred {p destination==
            :relative "on"
            :num_preds 1
            :pred {p object==
                     :color "green"
                     :object "table" } } }
```

Figure 3-3: SLS-Lite semantic frame for sentence "Put the blue box on the green table."

```
{c put
   :topic {q object
             :name "box"
             :pred {p color
                      :name "blue" } }
   :pred {p relative
            :name "on"
            :topic {q object
                      :name "table"
                      :pred {p color
                               :name "green" } } }
```

Figure 3-4: Conventional semantic frame for sentence "Put the blue box on the green table."

of the utterance contain useful information which should be kept in the frame. Given the developer's description of the domain, we had to generate several sets of TINA rules: one to initially discover which keys were used in sentences; a second to learn the structure in any example sentences; and a final grammar which is used in the actual interaction with the user. This process is described further in Chapter 5.

3.1.7 Discourse and Dialogue Management Server

GALAXY can make use of a dialogue management server to control the interaction between the user and the system until it has enough information to send a full query to the back-end, which can in turn do higher level dialogue direction [14].

In this initial version of SLS-Lite, we have not used a dedicated dialogue management server. Instead, we chose to provide a history mechanism within the back-end, and allow the developer to provide all dialogue management. This makes both the SLS-Lite system itself and the developer's interface to it much simpler, but makes it more difficult for the back-end developer by forcing them to manage the discourse themselves. In future versions of SLS-Lite, additional dialogue management may be done by the internal SLS-Lite systems, taking some of the burden off of the developer. More about SLS-Lite's discourse mechanism is described in Section 4.6.

3.1.8 Language Generation Server

The language generation server provides a method for converting the internal semantic frame representation into one of several natural (like English) or formal (like SQL) languages [1]. These responses can be used as further queries within a system, or can be sent to the user as responses.

We used MIT's GENESIS-II [1] server to take the semantic frame generated by TINA and, using SLS-Lite's built configuration files, create appropriate CGI arguments to pass to the developer's back-end. We did not use it to generate English sentences, since the application itself had the responsibility of generating the full replies, although, in the future, it is possible that SLS-Lite could generate the responses itself.

3.1.9 Text-to-Speech Server

Many applications require the system to speak back to the user – especially over the telephone. The text-to-speech server can take a sentence and generate sound data of that string being spoken, which can in turn be played back to the user through the audio server.

For this task we used a DECtalk [5] server which is set up to work within the GALAXY framework. We used this server to generate the sounds for the English-language replies which the application's back-end provides.

3.2 SLS-Lite Interface

As described in Section 3.1, SLS-Lite made use of certain GALAXY components, like the speech recognizer, configuring them for the appropriate tasks. Other components we provided our own servers for, such as the application back-end. Certain components weren't necessary at all within SLS-Lite, and so were not used. Figure 3-5 shows an overview of the GALAXY setup we used in SLS-Lite.

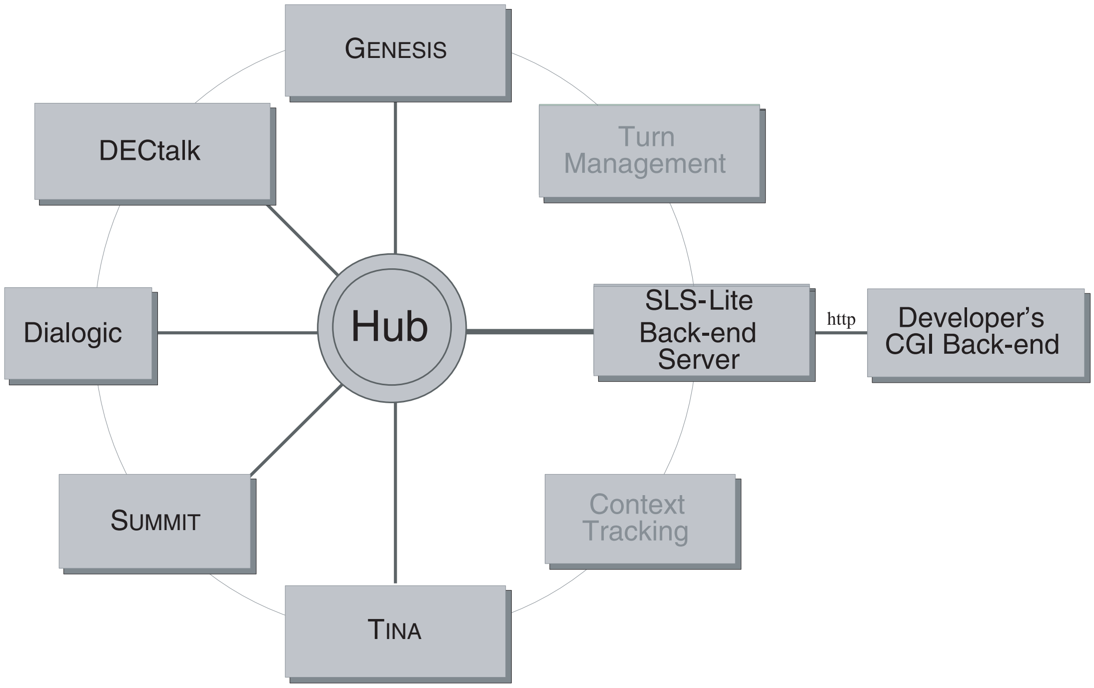

Figure 3-5: Overview of GALAXY-II setup used in SLS-Lite.

Many decisions were made along the way in determining how SLS-Lite would work, and what the internal and external setups would be. In this section, we will discuss some of these decisions about the architecture of SLS-Lite, and why we chose to do them the way we did.

3.2.1 Web-Based Interface

Since the ease of use for developers is very important to SLS-Lite, we had to choose an appropriate platform for the interface. Although a native C program and a proprietary client-server architecture might be more powerful, they would have required a much longer development time and be significantly less portable than the alternatives. A Java client within a web browser would be easier to build, and have much of the interface power of a full-fledged native application. However, it would still require a proprietary client-server protocol to communicate the data from the developer to the system and tell the system to perform functions such as building and running domains In addition, it would require a Java-enabled browser to use.

In the end, we decided on a purely HTML interface presented from a CGI script. This meant limiting the amount of interactivity (and hence, the power) of the developer's interface. However, it also meant that we could develop the interface much more rapidly. No proprietary protocols were necessary, since all changes were done on the server side. Further, it gave us a maximum amount of portability. Any web browser which supports forms and tables (which includes virtually every modern browser) can use SLS-Lite, simply by entering its URL.

3.2.2 CGI Back-End

Rather than have every developer write a GALAXY-compliant back-end, we decided to write a generic GALAXY back-end server which would, in turn, communicate to the developer's application through a web server, using the standard CGI (Common Gateway Interface) mechanism. This way, a developer could build an application with an existing set of tools using one of many languages such as C, Perl, or Python.

3.2.3 Keys and Actions

We decided to break the data provided by the developer into two parts – "keys" and "actions". Both keys and actions are referred to as "classes" because they each represent a set of words or phrases which can be grouped under a single name.

Keys are the concepts that are important to the meaning of a query. They cover the individual pieces of spoken information that the back-end needs to know about. They carry most if the actual information about what the user said. For instance, in the house domain there might be a "room" key which contains "living room", "dining room", and "kitchen". Each set of keys is given a name, and is considered equivalent in the SLS-Lite grammar. This means that, if "Turn the lights in the kitchen on," is an acceptable sentence, so is, "Turn the lights in the dining room on."

Any keys which occur in the sentence are reported to the back-end, meaning that the system will get as many relevant pieces of information as the user provides. Using this mechanism allows the developer to specify whole classes of information which are relevant to the conversation, without having to mark them every time they occur, or list every possible word which is valid in a given context.

Actions are complete sentences which are provided by the developer in the form of examples, which the SLS-Lite system generalizes. The developer groups the examples into action classes, so that each action class contains only examples which are similar in nature. For instance, in the house domain, the "turn" class might contain all of the examples of a user asking the system to turn something on or off. It might contain, "Turn the lights in the kitchen on," "Can you please turn off the dining room lights," and a multitude of other examples. For every utterance spoken by the user, a single action is reported to the back-end. This action represents whatever example matched the utterance. This allows the developer to choose groups of queries which are similar in nature and should be reported to the back-end under the same category.

The actions and keys of a given sentence go through several steps and conversions before reaching the form used by the CGI back-end. These steps are described in more detail in Sections 3.1.6 and 4.5.

3.2.4 XML Format

In addition to being able to generate the necessary data formats for all of the GALAXY components, SLS-Lite needed its own format for storing internal descriptions of the SLS-Lite domains. Rather than introduce another proprietary format, we decided to use an XML-based format for keeping track of the data in internal domains.

XML allows for very flexible specifications, ensuring that we would be able to add new capabilities to SLS-Lite without having to redesign and convert our file formats. XML is a well-known and publicly available format [21], meaning that external developers wishing to modify the data can learn how to do so with minimal effort. Finally, there are many publicly available tools for reading and manipulating XML data, making it easier for us as well as for external developers to read, understand, and modify SLS-Lite data.

Examples of the XML-derived data format used for SLS-Lite can be found in Appendices C, D, and E.

Chapter 4

Language Understanding

The most important part of the development of SLS-Lite was the decision of how the language understanding aspects should be built. Certain features of how SLS-Lite learns from the developer and generates grammars were determined by our desire to make things easy and flexible for the developer (even when they are not easy in the underlying system), while others were important to maintain the robustness of the final system.

4.1 Structured Grammar

In SLS-Lite, we chose to use a fairly structured grammar, allowing for (but not requiring) hierarchy. This provides a flexibility that a strictly flat grammar cannot achieve.

For instance, in the toys domain, the example "Put the blue sphere on the table behind the red box," would be difficult to convert to a flat meaning representation. In a flat grammar, such as a keyword spotter, there would be no way to differentiate which colors related to which objects, and what different roles the various objects in the sentence play semantically. In our system, the structured grammar allows the developer to break that sentence down into phrases, so that we know that the color blue and the object sphere are related, as well as to name various constituents, so we know that the blue sphere is a source and the table is a destination. More on how hierarchy is used in SLS-Lite is described in Section 5.3.

4.2 Robust Parsing

We designed the SLS-Lite system so that, when it is unable to fully parse a sentence (because the user said something which was not specified by the developer), it will fall back to robust parsing. This means that it will essentially become a phrase-spotting grammar. It will attempt to find all of the keys which it can in the utterance, and report them to the back-end. Since the action is based on which example matched the utterance, SLS-Lite reports an action of "unknown" for sentences that were robustly parsed. For instance, a sentence fragment received from the recognizer might parse as shown in Figure 4-1. TINA is able to determine some of the structure of the fragment, but does not know which action to use, since it was unable to generate a full parse.

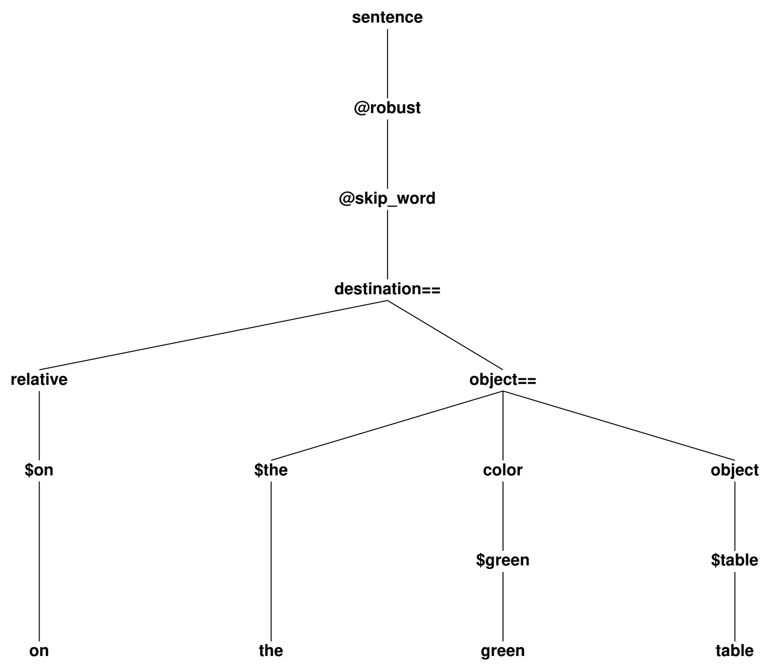

Figure 4-1: TINA robust parse tree for for sentence fragment "on the green table".

This feature gives SLS-Lite an added layer of robustness. Until speech recognizers are perfect and people are consistent, there will always be utterances which are misrecognized or formulated by the user in a way which was not planned for by the developer. With the robust parsing mechanism, when the recognizer or natural language component doesn't fully understand a sentence, the back-end can respond intelligently. For instance, if the user either said, or the recognizer misunderstood an utterance as "for Jef Pearlman", the back-end would now be able to react reasonably by saying, "What did you want to know about Jef Pearlman?".

4.3 Example-Based Generalization

After the developer has defined a few sets of concepts (keys), SLS-Lite learns nearly everything else from a set of examples. If the developer chooses to make the domain hierarchical, then only a few examples of the hierarchy must be explicitly bracketed by the user to show the hierarchy. From these examples, SLS-Lite will attempt to find the hierarchy in all of the other sample sentences that the developer provides.

While discovering a grammar based on generalizing from examples may not produce a grammar which is as robust as one which is hand-created by an expert, the goal of SLS-Lite was to enable non-experts to create these systems. Generalization makes grammar creation exceptionally easy – all the developer needs to do is pick out which words are important for the back-end to know about, and give examples, in plain English, of the ways in which a user might formulate a query.

4.4 Hierarchical n-Gram Language Model

When generating the language model for the recognizer to use, we chose a "hierarchical n-gram". While this is more complex than other n-gram grammars (such as a word class n-gram), this complexity is hidden from the developer and user, and the choice offers other advantages.

4.4.1 Word n-Gram

An n-gram is a very commonly used statistical language model, so named because it is built on recording the probability of any given word occurring, given the previous

$n-1$ words. This makes it much more flexible than a context free grammar like that used in the VoiceXML format described in Section 2.1. In a context free grammar, every possible query construction needs to be directly enumerated, making it very fragile for speech recognition. Instead, the n-gram model results in the probability of a specific word being calculated for each and every set of $n-1$ previous words:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} P(w_i | w_{i-1}, \ldots, w_{i-(n-1)})$$

While this does not seem to model the structure of natural language, it has proved to be an effective and competitive model in past and current speech systems, such as MIT's SUMMIT [6] recognizer.

4.4.2 Word Class n-Gram

One improvement often made the the word n-gram model is to replace actual words with equivalence classes of words. This "word-class n-gram" model makes use of the fact that many words are essentially equivalent in terms of their functions, and replaces the probabilities calculated on those words with a single set of probabilities for the equivalence class:

$$P(w_1, \ldots, w_N) = \prod_{i=1}^{N} P(w_i | c_i) P(c_i | c_{i-1}, \ldots, c_{i-(n-1)}) \text{ where } w_i \in c_i$$

A good example of this is cities – rather than calculate the probability of "Newark" and the separate probability of "Boston" given the previous $n-1$ words, the system would calculate the probability of the "city" class (containing both Newark and Boston) given those previous $n-1$ equivalence classes (which may be single words). A simple unigram is used to determine the probability of a word occurring given a particular class. Using a word n-gram has the result of reducing perplexity and often improving recognition.

4.4.3 Hierarchical n-Gram

A further improvement which has been made is the use of the phrase class n-gram model [8]. In this model, limited amounts of regular structure, like dates and times, can be interpolated into an n-gram. Thus, "the thirteenth of July", can be modeled as a unit in the same class as "the seventeenth of November", resulting in better, more accurate models.

Since a word class n-gram does not deal with multiple words which are really a single unit (such as San Francisco), speech system architects have generated artificial "underbar words", like "San_Francisco". This makes the two words into a single probabilistic unit. With phrase class n-grams, this is unnecessary, since San Francisco can be a single member of an equivalence class.

In SLS-Lite, we used a slightly different model, known as a hierarchical n-gram. One of the important issues was that MIT's phrase class n-gram tools are unable to deal with ambiguity, since they must first reduce the sentence into its component classes before determining the probabilities. This means that if the user asks a question about "Metallica", where Metallica is both a band name and an album name (two separate classes), the phrase class n-gram recognizer will fail because it is unable to determine which class it should reduce Metallica to. The hierarchical n-gram model we use will be able to consider both cases, and choose the class which has the highest probability within the model.

The hierarchical n-gram model also lacks the reliance on underbar words. This means that neither SLS-Lite nor the developer has to worry about building artificial underbar words, which results in a simple, flexible interface.

Finally, using the hierarchical n-gram provides a consistent language model for both the speech recognizer and the natural language understanding components of the final system, because the hierarchical grammar developed from the example sentences is used for both recognition and understanding. This makes the system much more robust, because it is unlikely that the recognizer will assign a high probability (and therefore choose) a hypothesis which the natural language component cannot parse.

Since both systems are given models which were generated from the same data by the same system, and have similar structures, high-probability output from the recognizer will be more likely to contain structures which will be understood by the natural language component.

4.5 Meaning Representation

We used the GENESIS-II [1] server to convert TINA's semantic frame into a meaning representation appropriate for passing as CGI arguments. GENESIS uses an internal representation which parallels the semantic frame structure used by TINA. Each class, which can contain a mixed set of words and other classes, functions as a clause, a predicate, or a key. In the SLS-Lite system, these correspond to actions, hierarchical subphrases, and keys, respectively, which were introduced in Section 3.2.3

Each clause is a full sentence, or action. Since there is only one full utterance per CGI request, the clause is passed as the first CGI argument, "action". For instance, for the sentence "Please put the blue box on the table," within the toys domain, the first parameter would be "action=put". If a robust parse occurred as described in Section 4.2 (and no user-defined clause was found), the parameter is set to "action=unknown".

GENESIS then takes the structure within the main clause and generates a second parameter, called "frame". Each key and predicate is included, along with its value, separated by commas and enclosed on the outside by parentheses. In order to have the entire frame structure be kept within a single CGI argument, we could not use "=" to separate them. Instead, we used "%3d". This code is translated to an "=" automatically by the CGI script, making it easier for the developer to read and parse the frame. In addition, spaces are replaced with the "+" symbol in order to be passed as CGI parameters. Like the "%3d", these are translated automatically to spaces by the CGI script.

In the case of a key, this process would produce something like "color=blue". In the case of a predicate, which signifies deeper hierarchy, a subframe in another set of parentheses represents the value: "source=(color=blue,object=box)". This results in a meaning representation which can be passed to a CGI script – in this case: "action=put&frame=(source=(color=blue,object=box),destination=(object=table))". If a robust parse occurred, as in the example in Figure 4-1, then as much information as TINA was able to discover will be stored in the frame, resulting in: "action=unknown&destination=(relative=on,object==(color=green,object=table))". More about this structure is discussed in Section 5.4.

The third CGI parameter, "history", is explained in the next section.

4.6 Discourse Mechanism

Since SLS-Lite has no internal discourse management, and CGI scripts are stateless (are re-run from scratch for each utterance), we needed to provide the developer with a way of managing focus. This meant that the CGI script needed some way to keep track of the current state of the conversation between utterances (each of which generates a separate CGI call). This gives much more flexibility to the developer, who can make a system which remembers what it or the user was talking about previously.

In order to keep the interface for the developer consistent, we chose to simply make the history mechanism into a string which the CGI server gives to the GALAXY back-end server, and is then returned as a CGI parameter called "history" with the next utterance of the same conversation. This provides a very simple, easy-to-use interface which can be extremely flexible as well.

For instance, one recommended use of the history parameter is to allow the CGI script to keep a structure representing the current state of the conversation. This can then be encoded into a frame structure similar to that for "frame", and when it is received later, decoded into the script's internal representation for use and modification. (For instance, if user asked the LCS-Info domain "What is the phone number for Jef Pearlman?", the script might generate a history line such as "(name=Jef Pearlman, property=phone)", so that when the user then asked "What about his email address?", the script would remember that the last name in focus was Jef

Pearlman. The GALAXY back-end server automatically takes care of appropriately encoding spaces and equals signs for CGI.

Chapter 5

Building a Domain

In this chapter, we will describe the step-by-step process of actually using SLS-Lite to make a new domain.

5.1 Web Interface

The web interface to SLS-Lite has several sections, all of which are presented on a single web page, and each of which will be discussed individually below. Figure 5-1 shows a fully expanded SLS-Lite editing window.

5.1.1 Domain Selection

The developer is initially presented with a set of buttons allowing them to add, remove, copy, and rename domains, as well as to select a domain to edit. Next to the buttons are a drop-down list of all existing domains, and a box where the developer can type a name for a new domain or for copying or renaming exiting domains. This domain selection menu is shown in Figure 5-2.

5.1.2 Domain Editing

Once a domain is selected for editing, a summary of that domain, like that in Figure 5-3, appears at the top of the page. Whenever the developer is editing a domain, this

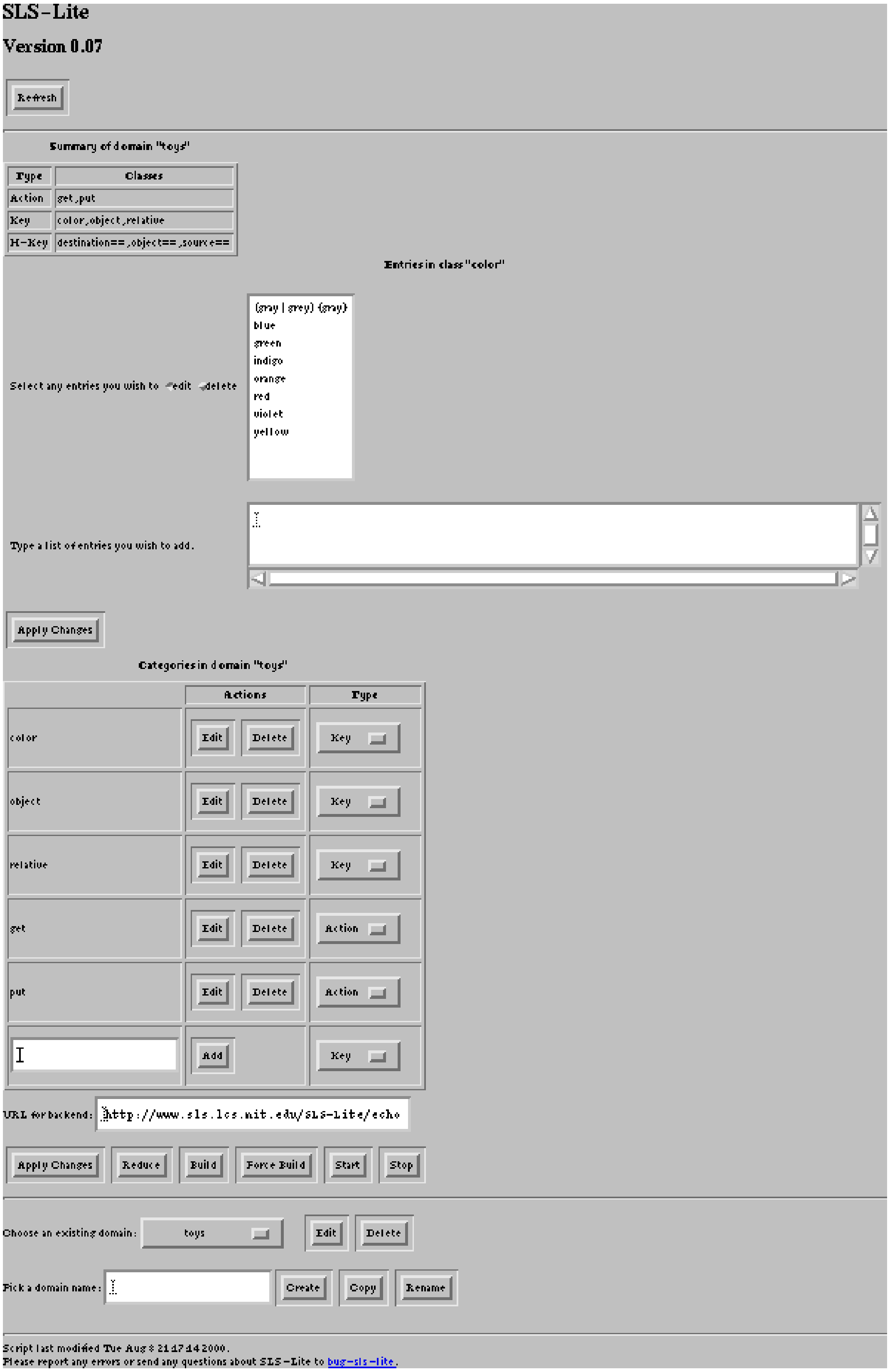

Figure 5-1: An overview of editing a fully expanded SLS-Lite domain.

Figure 5-2: Domain selection menu, with the house domain chosen.

summary will appear. It contains the name of the domain, and three lists. The first two lists contain all of the names of the actions and keys which have been explicitly defined by the developer. The third contains a list of all of the hierarchical classes which have been automatically determined from the developer's bracketing. These are called "H-Keys", for "hierarchical keys", since they represent concepts of hierarchy in the domain. This list can help the developer spot mistakes, such as making typographical errors, or using two different names for the same set of hierarchy.

Summary of domain "toys"

Type	Classes
Action	get, put
Key	color, object, relative
H-Key	destination==, object==, source==

Figure 5-3: Domain summary for the toys domain.

SLS-Lite also presents the developer with a more detailed listing of all the classes in the domain. For each class, it tells you whether the class is a key or an action, and allows you to change its designation. It also gives "Edit" and "Delete" buttons for each individual class, allowing you to modify them (as described in Section 5.1.3) or delete them one-at-a-time.

At the bottom of the list of classes is a text box, with a corresponding "Add" button and drop-down box which allow you to name a new class, and add it as either a key or an action. The class editing list is shown in Figure 5-4.

Below the class list is a box like the one shown in Figure 5-5, where the developer can enter a URL for the domain's CGI-based back-end. Whatever the developer enters into this box will be automatically contacted by SLS-Lite whenever someone

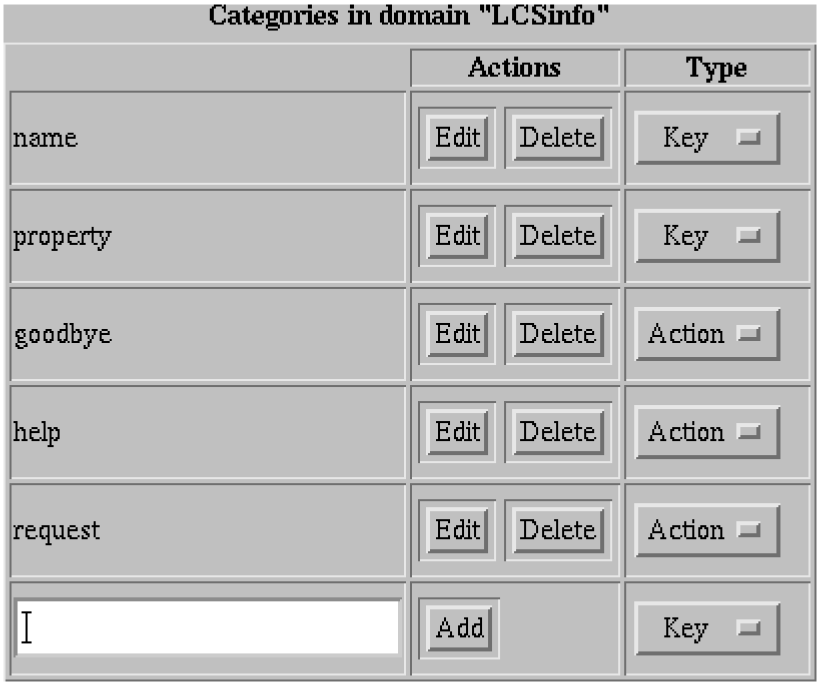

Figure 5-4: The class editing list for the LCS-Info domain.

uses the domain. By changing this entry, the developer can use a back-end located anywhere on the Internet, and even switch back-ends when necessary.

Figure 5-5: URL entry box for CGI back-end.

Below the URL box, SLS-Lite has a set of five buttons, shown in Figure 5-7. The first one, “Apply Changes”, makes any modifications made to the domain permanent. Other action buttons, such as those that add or edit classes, also make other changes to the domain permanent.

The “Reduce” button takes all of the example sentences given by the developer and simulates running them through the final system, showing a table containing the utterances and the CGI arguments which they would generate, as depicted in Figure 5-6. This allows the developer to debug the domain and make sure that all of their examples work as expected.

The “Build” button tells SLS-Lite to build all the necessary internal files to actually run the domain. These files are described in further detail in Appendix A for

Sentence Reduction	
Action	**Parenthesized**
goodbye	good bye action=goodbye&frame=()
help	give me help action=help&frame=()
	help action=help&frame=()
	how do i use this thing action=help&frame=()
	instructions action=help&frame=()
	please give me help action=help&frame=()
	please help action=help&frame=()
request	can you tell me the email address of jim glass action=request&frame=(name=Jim+Glass,property=email)
	can you tell me the title of victor zue action=request&frame=(name=Victor+Zue,property=title)
	do you know saman pl email address action=request&frame=(name=Saman+Amarasinghe,property=email)
	how about for jim glass

Figure 5-6: Partial sentence reduction for the LCS-Info domain.

the interested reader.

The "Start" and "Stop" button start and stop the actual human language technology servers for that particular domain. The domain which is run is configured as it was the last time the developer clicked the "Build" button – any changes since that point, while being retained by SLS-Lite, are not used in the actual running system. In the current SLS-Lite setup, the most recent domain to be run is the one the user is connected to when they phone SLS-Lite, although in the future, we will have several ways of allowing multiple simultaneous SLS-Lite domains to be run.

Figure 5-7: Buttons for apply changes to, reduce, build, start, and stop a domain.

5.1.3 Class Editing

Here we describe the interface for modifying classes through the SLS-Lite web interface. Particulars specific to editing keys or actions are described below, in Section 5.2.

Adding and Removing Entries

When the developer picks a particular class to edit, another section of the SLS-Lite interface appears. The class editor is identical for both keys and actions. As shown in Figure 5-8, the class editor contains two boxes. The first box lists all of the entries in the current class. The developer can select one or more existing entries from the list.

The second box is initially empty, and allows the developer to add entries. The developer can type one entry per line, and when changes are applied, these entries are made permanent and moved into the existing entry list.

Next to the existing list are a pair of radio buttons, marked "edit" and "delete". If the "edit" button is selected when the developer applies the changes, all of the selected entries are copied to the second box and removed from the existing list. This allows the developer to modify existing entries without having to retype them. If the "delete" button is selected, then the chosen entries are simply removed from the class.

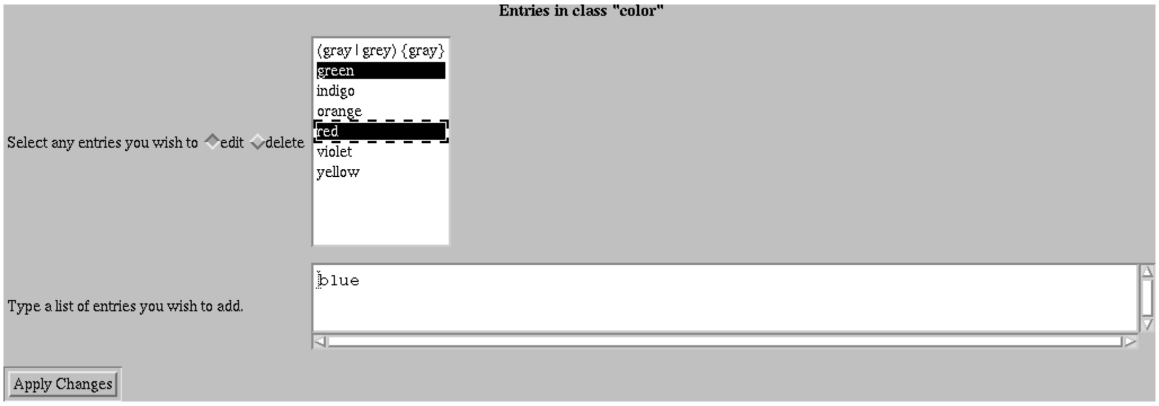

Figure 5-8: Editing the color key in the toys domain.

5.1.4 Using JSGF Markups

JSGF is the Java Speech Grammar Format [7], and was used briefly during the development of SLS-Lite. To simplify the creation of structurally similar sentences, we allowed the use of a subset of the JSGF specification within both keys and actions. Developers can enclose segments of a sentence to indicate that the enclosed words are optional, or can enclose several alternate phrases in a sentence within parentheses, separated by vertical bars.

For instance, a developer could enter the sentence, "[please] (put | place) source=(the red box) destination=(on the table)." SLS-Lite would automatically treat this as if the developer had entered all four variations of the sentence, correctly bracketed:

put source=(the red box) destination=(on the table)
please put source=(the red box) destination=(on the table)
place source=(the red box) destination=(on the table)
please place source=(the red box) destination=(on the table)

The additional use of the {} JSGF markup characters for keys is described below, in Section 5.2.

5.2 Keys and Actions

Keys determine what pieces of information are looked for by TINA and sent to the back-end inside the frame. Generally, all of the entries of a single key class should have the same function in an utterance, and should be of importance in what the user is going to say.

For instance, in the house domain, we wish to know what room the user is asking about, and all of the room names are interchangeable within a sentence, so we might want to create a "room" class, containing a list of all of the available rooms. To do this, we would simply add a new class of type key, named "room" (as described in

Section 5.1.2, and add to that class a list of rooms, one-to-a line, as described in Section 5.1.3.

In addition to the standard JSGF markup characters, keys allow the use of braces to enclose an "output form". Normally, the exact words which a user speaks are transmitted to the back-end if they match a key. For instance, in the LCS-Info domain, if "Jeffrey Pearlman" and "Jef Pearlman" are valid keys in the name class, and users ask both "What is the phone number for Jeffrey Pearlman?" or "What is the phone number for Jef Pearlman", the back-end will receive "name=Jeffrey Pearlman" and "name=Jef Pearlman" in the respective frames – that is, they will receive different keys despite the fact that they represent the same thing.

However, when the output form used, all of the forms of the key on the line containing the braces will, when inserted into a frame for the back-end, be rewritten as the phrase in brackets rather than the actual words spoken by the user. Instead of the above example, a line in the person class might read "(Jef | Jeffrey) Pearlman [P E A R L M A N] {Jef Pearlman}". The "{Jef Pearlman}" part of the line tells the system that, regardless of whether the user says "Jeffrey Pearlman P E A R L M A N" or "Jef Pearlman", the back-end should receive a parameter "name=Jef Pearlman".

This ability gives the developer added flexibility and makes the back-end easier to build. First, it allows him or her to build a simpler back-end which does not need to know about all of the variations on an input which have the same meaning. Second, it gives the developer the ability to do simple translations easily, such as converting spoken numbers to digital ones or word sequences to codes. For instance, in recognizing the phone number, the developer might add translations like "one {1}" to ensure that the back-end receives an actual numerical phone number. Similarly, an entry in a domain about flight planning might read "Boston {BOS}", so that the back-end received airport codes instead of actual city names.

In order to deal with ambiguous keys, it is possible to force SLS-Lite to treat a set of words in an action as having originated from a certain key. For instance, if "Metallica" is both an artist and an album, and an example reads, "Tell me what songs were sung by Metallica", then the SLS-Lite will, by default, choose either the album key or the artist key arbitrarily – and possibly incorrectly. In cases like this, the developer can specify which case to use by enclosing the words in less than and greater than signs, and placing the name of the class in front of the less than sign, followed by "=". In the above example, "Tell me what songs were sung by artist=<Metallica>" would force SLS-Lite to treat Metallica as an artist when trying to generalize that particular sentence.

Occasionally there will be words which the developer wishes a word or phrase to be a part of a class, but in certain instances wishes the same word or phrase to not be reduced to a key. For instance, the word "may" can be used both as a month and as a verb. In the sentence, "I would like a flight on May third," the developer would want the system to realize that May is a month, whereas if a sample question, "May I get a flight from Boston to Dallas," was asked, the developer would prefer to leave the word may alone.

To accomplish this, the developer can simply use two different capitalizations of May. Since SLS-Lite is case sensitive, "May" can be put in the month class, and "may" can be left out. Then, if the developer used the example sentences "I would like a flight on May third," and "may I get a flight from Boston to Dallas", the system will correctly generalize May in the first case only.

It is also useful to note that since the action variable is determined entirely by which action class matched the utterance, and all of the rest of the CGI arguments are determined from the keys and the back-end itself, there is no need for the {} markup characters in the actions.

5.3 Describing Hierarchy

As described in Section 4.1, SLS-Lite allows the developer to build a structured grammar where appropriate. To do this, the developer needs to bracket and name parts of some of the actions, in order that the system may learn where the structure lies. It is important to note, though, that if the developer chooses to bracket a sentence, he or she must bracket the *entire* sentence, since SLS-Lite will treat it as the complete description of that example. Also, SLS-Lite only generalizes hierarchy within a particular action. This means that if two separate actions exhibit similar hierarchical structures, at least one sentence in each class must be fully bracketed by the developer.

To bracket a sentence, the developer encloses the substructure which they wish to separate in parentheses, preceded by a name for the substructure followed by either "=" or "==", depending on whether the developer desires to use flattened or strict hierarchy, both of which are described in more detail below. Bracketing the hierarchy automatically results in SLS-Lite creating new hierarchical nodes in TINA's parse tree, so TINA will later understand it when parsing an utterance.

For instance, revisiting the example sentence from Section 4.5, "Please put the blue box on the table," the developer could bracket the sentence as, "Please put source==(the blue box) destination==(on the table)." Hierarchy can also be more than one level deep, and can mix both types of hierarchy. If the sentence was slightly more complex, it might read, "Please put source==(the blue box) destination==(on the table in the location=(kitchen))." Note that bracketing a sentence only involves pointing out the hierarchy – the keys are still automatically discovered by SLS-Lite.

5.3.1 Strict Hierarchy

The developer can specify strict hierarchy by using the "==" in bracketing a subsection of text. When strict hierarchy is used, all of the keys under the bracketed region are treated as they normally would, and each key becomes a key=value pair within the subframe, as described in Section 4.5. This provides a consistent means to bracket subsections, yet have each subsection retain the same keys and values as the developer would expect in a flat grammar. It also provides increased flexibility, since multiple levels of recursion will generate a consistent structure which is easy for the back-end to deal with.

For instance, if the sentence above, bracketed as "Please put source==(the blue box) destination==(on the table)," were spoken, its frame would look like "source=(color=blue,object=box),destination=(object=table)". With an extra level of hierarchy, "Please put source==(the blue box) destination==(on the table in the location==(kitchen))," would become "source=(color=blue,object=box), destination=(relative=on,object=table,location=(room=kitchen))".

5.3.2 Flattened Hierarchy

It is not always desirable to receive all of the key/value pairs within a hierarchical section of the grammar. For instance, a query in a flight domain might read, "Book me a flight from Boston to Dallas." In this case, an example using strict hierarchy such as "Book me a flight from source==(Boston) to destination==(Dallas)," would result in a frame like, "source=(city=Boston),destination=(city=Dallas)". However, since the developer knows that sources and destinations are always cities, it might be simpler just to receive source and destination keys, without the nested city labels. Flattened hierarchy allows the developer to do exactly this.

When the developer specifies flattened hierarchy by using a "=" in bracketing a subsection of text, instead of the value of the subsection name being a set of key/value pairs enclosed in parentheses, the value will be composed of all of the keys inside the parentheses, separated by spaces. For the above example, we could bracket the sentence as "Book me a flight from source=(Boston) to destination=(Dallas)," and the resulting frame would be "source=Boston,destination=Dallas" – without any "city=" key/value pairs or parenthesized subsections. Thus, if the developer knows the type of key that will appear within a hierarchical subsection, he or she won't have to deal with parsing the key's name.

If more than one key is of the same value, it will separate those values by underbars. This allows developers to easily build parameters which are made up of more than one entry from a class, like a phone number, without having to dig through the key name for each entry. For instance, if there was a digit class holding the digits zero through nine, and the developer gave the example, "Who has the phone number number=(two five three seven seven one nine)?" SLS-Lite would return a frame containing "number=two_five_three_seven_seven_one_nine".

Although it might be possible to automatically determine which regions should be flattened by the number of concepts inside, we chose to let the developer specify the type of hierarchy they wanted so that they could be assured of exactly how the system would act.

5.4 Building a CGI Back-End

Once the domain has been designed, the developer needs to build the back-end which will provide the actual domain-specific interaction to the user. First, the developer needs to have access to a CGI-capable web server, and place the script to be used at a URL matching the one specified to SLS-Lite (see Section 5.1.2). Because of the flexibility of CGI, it doesn't matter whether the CGI back-end is actually a Perl script, a C program pretending to be a web server itself, an Apache module, or any other particular setup, as long as it adheres to the CGI specification. Since all of our testing was done using Perl and CGI.pm [3], we will use these to provide examples of how a back-end might be built. A simple example of a simple Perl CGI back-end is given in Appendix B.

The first thing the CGI script needs to do is produce valid HTTP headers. Most CGI packages should provide the ability to do this easily.

The action, frame, and history, are all passed as individual CGI parameters. The first parameter is "action", which simply tells which action matched the user's utterance. If no actions matched, and SLS-Lite wasn't able to fully parse the sentence, this variable is set to the string "unknown". When SLS-Lite first receives a new call, it sends the action "###call_answered###" so that the back-end can welcome the user to the domain.

The second parameter is "frame". This contains information on which keys were found in the parsing of the utterance. If the domain is hierarchical, it may have several levels of hierarchy built in. Unless the domain is very simple (perhaps containing one key per utterance), this variable is very difficult to use in its default form. The back-end will probably want to parse the frame (which has a fairly regular structure) into some internal representation. In our example domains, we built a Perl function to parse a frame and return a hash tree containing the structure of the frame. This means that at any level in the hash tree, the keys were all of the key types that were found, and the values were either the value of the key (if it wasn't hierarchical) or another hash table containing the inner context (if it was hierarchical). This makes it very easy to check whether specific keys exist in the frame, or to extract hierarchical information without trouble.

Generally, the back-end should first check which action was given. If the action is "unknown", then the script can either attempt to get some information out of the keys in the frame, or simply ask the user to try again. After the back-end has decided which action it is dealing with, it needs to check the appropriate keys. In some cases, the same action can use multiple sets of keys. For instance, in the house domain with the action "turn", a room may or may not be present, depending on whether the user said, "Turn the lights in the kitchen on," or "Turn all the lights on." The script can simply check for the existence of certain keys to determine which form was used, and take the appropriate action.

To tell SLS-Lite what to say to the user, the back-end needs only to print the English-language sentence, which will in turn be taken by the CGI mechanism of the web server and sent to the GALAXY back-end server. This reply can only be one line long, and must end with a carriage return. However, that line can be essentially as long as the developer wants, and can contain multiple sentences. In order to end a call, simply prefix the final response with "###close_off###", and SLS-Lite will hang up after speaking the last sentence.

To use the history mechanism, the script should simply print a line starting with "history=". The line can contain any data the developer wants to remember, and must also end with a carriage return. Further, the history line must occur *before* the line to be spoken, since everything after that is ignored.

The way we used the history mechanism in our test scripts was to parallel the frame structure. In addition to a Perl function to parse the frame, we created a function which would take a hash tree structure (like that generated by the frame parser), and produce a single-string history frame. This allows the back-end to make changes to such a structure to keep track of the current focus (such as who the user asked about last). The script can then encode this into a history string, and when it is received by the next call of the script, decode it back into the structure it started with.

By doing all this, a script can have a fairly complex interaction with a user, understanding what the user requests, responding appropriately, and keeping track of the course of the conversation as it goes – all using some very simple mechanisms to interact with the main SLS-Lite system.

Chapter 6

Conclusions

This thesis has discussed the development of the SLS-Lite system and many of the issues which we faced during its design. We have covered both the architecture of the system and the language understanding aspects of the work.

The work done on SLS-Lite differs from previous work in several ways. It concentrated on mixed-initiative type dialogue, whereas most development in this area to date has been on directed dialogue. It provides a full speech system to developers, where they need only set up the domain's grammar and build a simple CGI-based back end to run it. Finally, it allows the construction of a robust, flexible, hierarchical grammar with a very simple interface, usable by experts and novices alike.

There are several directions in which future work on SLS-Lite might proceed. Some of these are discussed below.

6.1 Discourse and Dialogue

SLS-Lite concentrated mainly on simplifying the language modeling aspects of building speech systems, and left the dialogue management to the back-end developers. Since GALAXY has provisions for automated dialogue management, future work could involve automating the configuration of such systems.

If SLS-Lite were to provide a simple way to do some automated dialogue management, then some of the burden could be taken off of the back-end. For instance, the

LCS-Info domain might automatically keep track of which person and what property was being discussed, automatically asking for more information where necessary, and filling in incomplete queries with information retained from previous interactions.

6.2 Ambiguity

Some English sentences are ambiguous. This means that, even to native speakers, they do not have a single "correct" parse. For instance, in the sentence, "Put the blue box on the red table behind the green block," it is unclear whether the green block is on the red table the blue box should be placed behind it, or if the red table is itself behind the green block.

Currently, SLS-Lite chooses a single parse which is the most likely in its model, and reports that to the back-end. In the future, SLS-Lite could report all valid parses of the sentence in multiple frame arguments, and allow the back-end to decide which, if any, to use. Much like the robust parsing mechanism, this will give the back-end added flexibility when SLS-Lite isn't able to fully and unambiguously parse an utterance.

6.3 Architecture

SLS-Lite currently only allows one domain to be executed at a time. In order to fully make use of the system and allow as many people as possible to use it, future developers could modify the system so that multiple domains can run simultaneously. This will become especially important as the group of people developing domains diversifies. As the set of developers spreads beyond a small group at MIT, there will be less interpersonal coordination, and the management of domains will have to become more automated in order for multiple developers, users, and domains, to work on the same system independently of each other.

6.4 Developer Interface

Several improvements could be made to the way the developer interacts with SLS-Lite.

6.4.1 Java Interface

A Java-based interface to the SLS-Lite page could offer far more interactivity and power than the current interface. Apart from the purely aesthetic improvements, most domain changes could be made in real-time, without reloading the page. Also, editing of classes could be much more efficient, without the need to move text back and forth between lists in order to edit it.

6.4.2 Security

Currently, SLS-Lite has no security model. Anyone can modify or delete (although deleting makes a backup) anyone else's data. This is acceptable when the users of SLS-Lite are limited to those within our research group, or at least within MIT, but if SLS-Lite is to be opened to those outside MIT, some sort of user verification will have to be put in place. This can be something as simple as a web server's user verification, or an actual SSL connection with SLS-Lite maintaining its own private access control lists.

6.5 User Interface

In addition to improvements in the developer interface, several improvements could be made in the user's experience of SLS-Lite. As the number of domains grows, the prospect of all developers and domains sharing a single phone line becomes less acceptable. Below are some possible ways SLS-Lite could be modified to alleviate this problem.

6.5.1 Call Redirector

Rather than have the SLS-Lite phone number dial directly into whatever domain is running, we can have an automated (possibly SLS-Lite based) system which answers the phone, asks the user what domain they wish to speak to, and automatically transfers control to the appropriate SLS-Lite domain. The GALAXY architecture has the capability of running such a system – SLS-Lite would simply have to configure it appropriately each time a domain was added or removed.

6.5.2 Non-Telephone Interface

Another solution which could work separate from or in addition to the call redirector is to make use of our native Linux and Windows clients to allow users to talk to SLS-Lite systems directly from their machines. Over the course of this thesis, we also developed a Java-based audio server, which used native libraries for actually recording and playing back sound data. Use of Java-based clients would allow a much broader range of client architectures to be used with minimal effort.

Further, with the release of the Java Development Kit version 1.3, Java will have cross-platform sound, allowing a single Java client to be used on any system which has a recent JDK. This means that, if the newer versions of Java are incorporated into web browsers, SLS-Lite systems could even be accessed directly from inside web pages. This type of interface will allow even more users to talk to SLS-Lite systems, resulting in more training data and more development of the technologies.

Appendix A

SLS-Lite Configuration Files

In this appendix, we will briefly describe some of the configuration files for various GALAXY components which are automatically generated by SLS-lite. Where *domain* is used, it represents the name of the particular SLS-Lite domain in question.

***domain*.xml** the SLS-Lite native XML format for storing a full description of the domain.

DOMAIN_DEFS a set of definitions telling GALAXY servers where individual data files are located.

***domain*.tina** describes the TINA grammar used to parse utterances (both full and robust).

***domain*.constraints** a set of constraints to allow TINA to produce full parses in preference to robust parses by lowering the probability scores for the latter.

***domain*.translations** used by TINA to translate outputs to a consistent form – generated by {}'s from key class entries.

***domain*.sents** a full set of example sentences given by the developer, but stripped of hierarchy, used for automated training and reduction for the developer interface

***domain*.actions** used by TINA to convert a parse tree to a semantic frame.

***domain*.mes** used by GENESIS to generate CGI arguments from a semantic frame.

***domain*.rewrite** used by GENESIS to rewrite patterns to a desired format.

***domain*.slabels** used by SUMMIT for pronunciation dictionary generation.

***domain*.rec** used to control recognition process in SUMMIT.

***domain*.bcd** pronunciation dictionary searched by SUMMIT during recognition.

***domain*.bcd.fst** finite state transducer (FST) version of pronunciation dictionary.

***domain*.bigram.fst** FST version of bigram grammar used by SUMMIT.

***domain*.cat** catalog file used by GENESIS.

***domain*.clg.fstb** FST file searched by SUMMIT containing composed dictionary and bigram grammars.

***domain*.baseforms** phonemic baseforms of word vocabulary file.

***domain*.blabels** labels associated with acoustic models.

***domain*.bmodels** acoustic models used by SUMMIT during recognition.

***domain*.fisher** Fisher discriminant models used for utterance-level confidence scoring by SUMMIT.

112.brec acoustic measurements performed by SUMMIT during recognition.

bpcs.trim miscellaneous file used for principal component analysis performed during recognition.

Appendix B

Example Perl CGI Back-end (Switches Domain)

Here we present a very simple domain, along with the XML data and an example CGI script which serves as the SLS-Lite backend. The domain lets the user change the state of two switches to on or off. The user can also set the state of one switch to that of the other, and query whether a particular switch is on or off or ask for the state of all switches at once. Actual examples of the different query types can bee seen in the XML class description below.

The switches domain uses both flattened and strict hierarchy. If this system were actually being deployed, the developer would most likely choose one type of hierarchy over the other (or use a flat domain), but in this case we used all three types of structure for demonstration purposes.

B.1 XML Data

Below is the actual XML data file for the switches domain.


```xml
<?xml version="1.0"?>
<slslite version="1.0" domain="switches">

<property SerialBuild="30"/>
```

```
<property SerialMod="30"/>
<property URL="http://sls-lite.lcs.mit.edu/SLS-Lite/switches.cgi"/>

<class name="ask" type="Action">
  <entry>is switch one On or Off</entry>
  <entry>which switches are on</entry>
</class>

<class name="good_bye" type="Action">
  <entry>good bye</entry>
</class>

<class name="number" type="Key">
  <entry>one {1}</entry>
  <entry>two {2}</entry>
</class>

<class name="onoff" type="Key">
  <entry>off</entry>
  <entry>on</entry>
</class>

<class name="set" type="Action">
  <entry>please set target==(switch one) to [be the same as] source=(switch two)</entry>
</class>

<class name="turn" type="Action">
  <entry>[can you] [please] turn switch [number] one on</entry>
</class>

</slslite>
```

B.2 SLS-Lite Perl Package

This is the source for the SLS-Lite Perl package we built and used, which provides functions to output responses (with or without history), parse frames into hash trees, and unparse hash trees into frames.

```
# SLSLite Package

# Contains functions for outputting responses and/or history lines,
# as well as parsing frames to hash trees and unparsing hash trees to
# frames.
```

```
package SLSLite;


# output
# Outputs a text string and an optional history line to the SLS-Lite
# backend server. Automatically strips extra carriage returns and adds
# one if needed, and adds the history= prefix.
#       SLSLite->output("Welcome to SLS-Lite.");
#       SLSLite->output("Welcome to SLS-List.", "(target=none)');

sub output {
    my $response = shift; # The response to speak.
    my $history = shift;  # The history variable to report, if any.

    # Print the history string if it was passed.
    if (defined($history)) {
        chomp $history;
        print "history=$history\n";
    }
    # Print the response string.
    chomp $response;
    print "$response\n";
}


# parse
# Parses a frame structure from the SLS-Lite CGI argument into a hash
# tree, returning a hash.
#       my %frame = SLSLite->parse(param('frame'));

sub parse {
    my $frame = shift;                     # The frame to parse.
    my @frame = split(/([(,=)])/, $frame); # The split up frame array.

    # Split the frame on parentheses, commas, and equals, and pass it to
    # the recursive parser, taking the resulting reference and returning
    # the actual hash table.
    return %{parse_recurse(@frame)};
}


# unparse
# Takes a hash tree and generates a valid frame, usually to be used as a
# history string and to be later parsed to regain the original structure.
#       my $history = SLSLite->unparse(%history);
```

```
sub unparse {
    my %data = @_;  # The hash structure to unparse.
    my $text = "("; # Start out with open parenthesis.
    my $first = 1;  # Flag to note that this is the first key at
                    # the current level (and so gets no comma before it).

    # Run through the keys, stringing them together with commas.
    foreach my $key (sort(keys(%data))) {
        $text .= "," unless $first;
        $first = 0;
        # If a key is a reference (hierarchical), recursively unparse it.
        if (ref($data{$key})) {
            $text .= "$key=".SLSLite->unparse(%{$data{$key}});
        } else {
            $text .= "$key=$data{$key}";
        }
    }
    # Return the final result, with closing parenthesis.
    return $text.")";
}


# parse_recurse
# Used recursively by parse and itself to parse frame. Do not call this
# function directly -- use parse.

sub parse_recurse {
    my @frame = @_;     # Split up frame.
    my %frame = ();     # Hash table of current level of structure.
    my $depth = 0;      # Depth within current level
                        # (0=outside, 1=inside, 2=substructure).
    my $lastkey = "";   # Previous key encountered (key=)
    my $lastword = ""; # Previous segment of frame.
    my @subframe = (); # List of segments contained in lower hierarchy to
                        # be recursively parsed.
    my $building = 0;   # Flag if we're inside recursive structure (and
                        # building @subframe).

    # Deal with each segment in turn.
    foreach my $segment (@frame) {
        # Skip blank segments.
        next if ($segment eq '');

        # If we hit a '=', record the previous segment as being our
        # key or h-key.
        $lastkey = $lastword if (!$building && $segment eq '=');
```

```
        # If we have reached a second open paren, then start recording keys at
        # the next level of hierarchy.
        if ($segment eq '(') {
            $depth++;
            if ($depth == 2) {
                @subframe = ();
                $building = 1;
            }
        }

        # If building a subframe, add the gurrent segment to the list to
        # be recursively processed at the closing parenthesis.
        push @subframe, $segment if ($building);

        # If we have a string (not open paren) and previous segment was '=',
        # we have a key/value pair to record.
        if (!$building && $segment ne '(' && $lastword eq '=') {
            $frame{$lastkey} = $segment;
        }

        # If we have hit the end of a segment, lower our depth. If that was
        # depth 1, we're done. Otherwise, we finished a recursive subframe,
        # and we need to recurse on it.
        if ($segment eq ')') {
            $depth--;
            if ($depth == 0) {
                return \%frame;
            } elsif ($depth == 1) {
                $frame{$lastkey} = parse_recurse(@subframe);
                $building = 0;
            }
        }

        # Record this segment as being the most recent.
        $lastword = $segment;
    }

    # Return a reference to our newly created (sub)frame.
    return \%frame;
}


1;
```

B.3 CGI Back-end Script

The following is an example CGI backend for the switches domain, written in Perl, and using the above SLSLite package.

```
#!/usr/local/bin/perl

# Include our SLSL support package.
use SLSLite;

# Use the CGI package.
use IO::Socket;
use CGI qw(:standard :html);
use CGI::Carp 'fatalsToBrowser';

# Print HTTP header.
print header;

# Decode the history frame if there is one, or initialize it if there
# isn't.
if (defined($history = param('history'))) {
    %state = SLSLite::parse($history);
} else {
    %state = (1 => 'unset',
              2 => 'unset');
}

# Decode the frame and retrieve the action.
%frame = SLSLite::parse(param('frame'));
$action = param('action');

# Choose what to do based on the action.
if ($action eq '###call_answered###') {
    # Respond to a new call.
    SLSLite::output("Welcome to the switch test domain. ".
                    "All switches are unset by default." );
} elsif ($action eq 'unknown') {
    # Robust parse. The easiest thing to do is to ask for the user to try
    # again.
    SLSLite::output("I'm sorry, I couldn't understand you.",
                    SLSLite::unparse(%state));
} elsif ($action eq 'turn') {
    # Retrieve the switch number and new state.
    $number = $frame{number};
    $onoff = $frame{onoff};
```

```
    # Do some error checking, for the example's sake.
    if (!defined($onoff) || !defined($number)) {
        # This should never occur in our current setup -- didn't get a number
        # or onoff in the frame.
        SLSLite::output("I'm sorry, you must specify both a switch and a state.",
                        SLSLite::unparse(%state));
    } elsif ($state{$number} eq $onoff) {
        # Check if it's already set, and report it if so.
        SLSLite::output("Switch number $number was already $onoff.",
                        SLSLite::unparse(%state));
    } else {
        # Change the state and report it.
        $state{$number} = $onoff;
        SLSLite::output("Switch number $number has been turned $onoff.",
                        SLSLite::unparse(%state));
    }
} elsif ($action eq 'ask') {
    # Check if we reported a switch number.
    if (defined($number = $frame{number})) {
        # Asking about a particular switch.
        SLSLite::output("Switch number $number is $state{$number}.",
                        SLSLite::unparse(%state));
    } else {
        # Asking about all switches. We are deliberately ignoring the
# onoff key here, although we could also use that to only
        # tell which lights match the onoff request, instead of just
# listing them all.
        SLSLite::output("Switch number 1 is $state{1} ".
                        "and switch number 2 is $state{2}.",
                        SLSLite::unparse(%state));
    }
} elsif ($action eq 'set') {
    # For the example, the target is strictly hierarchical while the source
    # is flattened.
    $source = $frame{source};
    # Now destination represents the "destination" subframe. We can treat
    # it exactly as if it were a full frame parse itself.
    %destination = %{$frame{destination}};
    $destination = $destination{number};

    if ($destination == $source) {
        # If the source and destination are the same, complain.
        SLSLite::output("Those are the same switch!",
                        SLSLite::unparse(%state));
    } elsif ($state{$destination} eq $state{$source}) {
        # If the source and destination are in the same state, complain.
```

```
        SLSLite::output("Switch number $destination is already $state{$source}, ".
                        "like switch number $source.",
                        SLSLite::unparse(%state));
    } else {
        # Match the states and report it.
        $state{$destination} = $state{$source};
        SLSLite::output("Switch number $destination has been turned $state{$source}, ".
                        "like switch number $source.",
                        SLSLite::unparse(%state));
    }
} elsif ($action eq 'good_bye') {
    # Say good bye and hang up.
    SLSLite::output("###close_off### Good bye.\n");
} else {
    # This is an unknown action -- should never occur.
    SLSLite::output("What?", SLSLite::unparse(%state));
}
```

Appendix C

House Domain

Below is the XML data file for the house domain.


```
<?xml version="1.0"?>
<slslite version="1.0" domain="house">

<property SerialBuild="11"/>
<property SerialMod="11"/>
<property URL="http://sls−lite.lcs.mit.edu/SLS−Lite/house.cgi"/>

<class name="good_bye" type="Action">
  <entry>good bye</entry>
  <entry>later</entry>
</class>

<class name="object" type="Key">
  <entry>(television | tv) {television}</entry>
  <entry>lights</entry>
  <entry>microwave</entry>
  <entry>toaster</entry>
  <entry>v c r {VCR}</entry>
</class>

<class name="onoff" type="Key">
  <entry>lit {on}</entry>
  <entry>off</entry>
  <entry>on</entry>
</class>

<class name="room" type="Key">
  <entry>dining room</entry>
```

```
  <entry>kitchen</entry>
  <entry>living room</entry>
</class>

<class name="status" type="Action">
  <entry>([can you] [please] tell me | do you know) (what | which) lights are on</entry>
  <entry>([can you] [please] tell me | do you know) if the (lights in the kitchen | kitchen lights)
    are on</entry>
  <entry>(is | are) the (dining room television | tv in the living room) On or Off</entry>
  <entry>(is | are) the (dining room television | tv in the living room) on</entry>
</class>

<class name="turn" type="Action">
  <entry>[can you] [please] turn all the lights off</entry>
  <entry>[can you] [please] turn off all the lights</entry>
  <entry>[can you] [please] turn off the (living room lights | lights in the living room)</entry>
  <entry>[can you] [please] turn the (living room lights | lights in the living room) off</entry>
</class>

</slslite>
```

Appendix D

Toys Domain

Below is the XML data file for the toys domain.


```xml
<?xml version="1.0"?>
<slslite version="1.0" domain="toys">

<property SerialBuild="25"/>
<property SerialMod="25"/>
<property URL="http://www.sls.lcs.mit.edu/SLS-Lite/echo.cgi"/>

<class name="color" type="Key">
  <entry>(gray | grey) {gray}</entry>
  <entry>blue</entry>
  <entry>green</entry>
  <entry>indigo</entry>
  <entry>orange</entry>
  <entry>red</entry>
  <entry>violet</entry>
  <entry>yellow</entry>
</class>

<class name="get" type="Action">
  <entry>get object==(the blue box) [source==(from the [red] table)]</entry>
</class>

<class name="object" type="Key">
  <entry>bed</entry>
  <entry>block</entry>
  <entry>box</entry>
  <entry>chair</entry>
  <entry>circle</entry>
```

```
  <entry>sofa</entry>
  <entry>table</entry>
  <entry>triangle</entry>
</class>

<class name="put" type="Action">
  <entry>put source==(the blue box) destination==(behind object==(the sofa)
    destination==(on object==(the green table) destination==(behind
    object==(the yellow table))))</entry>
  <entry>put source==(the red box) destination==(on object==(the [green] table))</entry>
  <entry>put the blue box here</entry>
  <entry>put the blue box on top of the green table</entry>
</class>

<class name="relative" type="Key">
  <entry>behind</entry>
  <entry>on</entry>
  <entry>on top of {on}</entry>
</class>

</slslite>
```

Appendix E

LCS-Info Domain

Below is the XML data file for the LCS-Info domain.


```
<?xml version="1.0"?>
<slslite version="1.0" domain="LCSinfo">

<property SerialBuild="734"/>
<property SerialMod="734"/>
<property URL="http://sls-lite.lcs.mit.edu/SLS-Lite/miner.cgi"/>

<class name="goodbye" type="Action">
  <entry>good bye</entry>
</class>

<class name="help" type="Action">
  <entry>[please] [give me] help</entry>
  <entry>how do i use this thing</entry>
  <entry>instructions</entry>
</class>

<class name="name" type="Key">
  <entry>(jim | james) glass {Jim Glass}</entry>
  <entry>Aaron Edsinger</entry>
  <entry>Aaron Isaksen</entry>
  <entry>Aaron McKinnon</entry>
  <entry>Aaron Solochek</entry>
  <entry>Abhi Shelat</entry>
  <entry>Abidemi Adeboje</entry>
  <entry>Adam Glassman</entry>
  <entry>Adam Klivans</entry>
  <entry>Adam Smith</entry>
```

```
  <entry>Adel Hanna</entry>
  <entry>Adrian Birka</entry>
  <entry>Agnes Chow</entry>
  <entry>Alan Edelman</entry>
  <entry>Alan Kotok</entry>
  <entry>Alantha Newman</entry>
  <entry>Albert Ma</entry>
  <entry>Albert Meyer</entry>
  <entry>Alcira Vargas</entry>
  <entry>Alejandra Olivier</entry>
  <entry>Alex Hartemink</entry>
  <entry>Alex Shvartsman</entry>
  <entry>Alex Snoeren</entry>
  <entry>Alex Wong</entry>
  <entry>Alexandra Andersson</entry>
  <entry>Alexandro Artola</entry>
  <entry>Alexandru Salcianu</entry>
  <entry>Alexey Radul</entry>
  <entry>Ali Tariq</entry>
  <entry>Alison Crehan</entry>
  <entry>Allen Miu</entry>
  <entry>Allen Parseghian</entry>
  <entry>Allison Waingold</entry>
  <entry>Amay Champaneria</entry>
  <entry>Amit Sahai</entry>
  <entry>Anand Ramakrishnan</entry>
  <entry>Anant Agarwal</entry>
  ⋮
  <entry>Yan Zhang</entry>
  <entry>Yao Sun</entry>
  <entry>Yevgeniy Dodis</entry>
  <entry>Ying Zhang</entry>
  <entry>Yuichi Koike</entry>
  <entry>Ziqiang Tang</entry>
  <entry>Zulfikar Ramzan</entry>
  <entry>saman [amarasinghe] {Saman Amarasinghe}</entry>
  <entry>victor [zue] {Victor Zue}</entry>
</class>

<class name="property" type="Key">
  <entry>(office [number] | room [number] | location) {room}</entry>
  <entry>(telephone | phone) number {telephone}</entry>
  <entry>(title | position) {title}</entry>
  <entry>email [address] {email}</entry>
</class>

<class name="request" type="Action">
```

```
  <entry>(what | how) about [for] jim glass</entry>
  <entry>can you tell me the email address of jim glass</entry>
  <entry>can you tell me the title of victor zue</entry>
  <entry>do you know saman pl email address</entry>
  <entry>i (would like | want) to know how to get to the office of jim glass</entry>
  <entry>tell me his room number [please]</entry>
  <entry>what about (his | her | their) office number</entry>
  <entry>what is the phone number for victor zue</entry>
  <entry>what is the title of saman amarasinghe</entry>
  <entry>what is victor zue pl phone number</entry>
</class>

</slslite>
```

Bibliography

[1] L. Baptist and S. Seneff. Genesis-II: A Versatile System for Language Generation in Conversational System Applications. In *Proc. ICSLP*, Beijing, China, October 2000.

[2] E. Barnard, A. Halberstadt, C. Kotelly, and M. Phillips. A Consistent Approach to Designing Spoken-Dialog Systems. In *Proc. ASRU Workshop*, Keystone, Colorado, 1999.

[3] CGI.pm - a Perl5 CGI Library, http://stein.cshl.org/WWW/software/CGI/.

[4] CSLU Speech Toolkit, http://cslu.cse.ogi.edu/toolkit/.

[5] DECtalk Speech Synthesis, http://www.forcecomputers.com/product/dectalk/dtalk.htm.

[6] J. Glass, J. Chang, and M. McCandless. A Probabilistic Framework for Feature-Based Speech Recognition. In *Proc. ICSLP*, October 1996.

[7] Java Speech Grammar Format Specification, http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/JSGF.html.

[8] M. McCandless and J. Glass. Empirical Acquisition of Language Models for Speech Recognition. In *Proc. ICSLP*, Yokohama, Japan, September 1994.

[9] Nuance, http://www.nuance.com/.

[10] C. Pau, P. Schmid, and J. Glass. Confidence Scoring for Speech Understanding Systems. In *Proc. ICSLP*, Sydney, Australia, November 1998.

[11] S. Seneff. Robust Parsing for Spoken Language Systems. In *Proc. ICASSP*, San Francisco, California, March 1992.

[12] S. Seneff. TINA: A Natural Language System for Spoken Language Applications. *Computational Linguistics*, 18(1), 1992.

[13] S. Seneff, E. Hurley, R. Lau, C. Pau, P. Schmod, and V. Zue. GALAXY-II: A Reference Architecture for Conversational System Development. In *Proc. ICSLP*, 1998.

[14] S. Seneff, R. Lau, and J. Polifroni. Organization, Communication, and Control in the GALAXY-II Conversational System. In *Proc. Eurospeech*, Budapest, Hungary, September 1999.

[15] SpeechWorks International, http://www.speechworks.com.

[16] Tellme, http://www.tellme.com/.

[17] Tellme Studio, http://studio.tellme.com/.

[18] Nuance V-Builder, http://www.nuance.com/index.htma?SCREEN=vbuilder.

[19] VoiceXML Forum, http://www.voicexml.org/.

[20] VoiceXML Specification Version 1.0, http://www.voicexml.org/specs/VoiceXML-100.pdf.

[21] Extensible Markup Language (XML), http://www.w3.org/XML/.

[22] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pau, T. J. Hazen, and L. Hetherington. JUPITER: A Telephone-Based Conversational Interface for Weather Information. In *IEEE Trans. SAP*, volume 8, January 2000.