

**Rapid Configuration of Discourse and Dialog  
Management in Conversational Systems**

by

Sterling J. Crockett

B.Sc., Massachusetts Institute of Technology (2000)

Submitted to the Department of Electrical Engineering and Computer  
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

© Massachusetts Institute of Technology 2002. All rights reserved.

Author .....  
Department of Electrical Engineering and Computer Science  
May 21, 2002

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

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



# **Rapid Configuration of Discourse and Dialog Management in Conversational Systems**

by  
Sterling J. Crockett

Submitted to the Department of Electrical Engineering and Computer Science  
on May 21, 2002, in partial fulfillment of the requirements for the degree of  
Master of Engineering in Electrical Engineering and Computer Science

## **Abstract**

A sophisticated conversational system should be able to carry a complex dialogue with the user and maintain a memory of information which has been learned. Although the framework to create such conversational systems does exist, known as the Galaxy architecture, it has not been available to less experienced developers through existing development tools. To address this issue, SpeechBuilder, a development utility, has been modified to be able to make use of dialogue and discourse. These changes to SpeechBuilder were made possible by the creation of a generic dialogue server, which enables developers to more easily configure dialogue. It also enables the use of external application servers, which further enhance the ability of system domains. A sports score information system has been built with SpeechBuilder which demonstrates that these functionalities have been added successfully.

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



## Acknowledgments

First, I would like to express my very deep gratitude to T.J. Hazen, Jim Glass, and Victor Zue for bringing me into the Spoken Language Systems group. This is an act for which I will always be grateful to them, and which I will never forget. After I joined the group, they continued to provide valuable support, feedback, and comic relief at the most appropriate times.

In a greater sense, my experience in SLS has been a transformation of attitude toward the academic community. My experience as an undergraduate was almost universally that professors and teaching assistants were cold and disinterested when I needed help. In stark contrast, the SLS group has been very warm, and staff members have even seemed to make time to help where it didn't exist. I have been constantly impressed by the outpouring of support from the time I joined, to when I had surgery, and finally in writing my thesis.

In particular, Joe Polifroni has been a wonderful person to work with for the last year; he is a master of feedback, and any person would be lucky to work with him. Without his help, I would never have been able to do what I have done with this project.

I would also like to say a special thank you to Michelle Spina, who read this thesis and gave me excellent suggestions.

As my thesis advisor, I could not have asked for a better person than Jim. I have heard horror stories of students who spent years working on thesis research, only to have it rejected at the last moment because their advisor had not taken time to make sure it met the standards of quality. Jim, however, has met with me consistently on a weekly basis. Even with all the new responsibilities and time commitments as group leader he has helped me keep on schedule, and has always had an extra minute or two to talk about hockey.

A special thanks goes to my office mates, Issam Bazzi, Min Tang, and Karen Livescu who have tolerated my unorthodox schedule, humming (and sometimes outright singing) with laudable patience.

My family does not know what my research has been, but they have loved me the same over the last 8 years since I left home. On the darkest of days, they have been an unmovable pillar of support.

Finally, and most importantly, I must thank my beautiful wife Julie. I cannot possibly thank her enough for her perpetually supportive attitude and caring nature. She has believed in me, she has always been my advocate in every situation, and I know that will never change.

This research was supported by DARPA under contract N66001-99-1-8904, by NTT and under an industrial consortium supporting the MIT Oxygen Alliance.



# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Galaxy . . . . .	12
1.1.1	Dialogue . . . . .	14
1.1.2	Discourse . . . . .	15
1.2	SpeechBuilder . . . . .	16
1.2.1	What is SpeechBuilder? . . . . .	17
1.2.2	Why is SpeechBuilder useful to developers? . . . . .	18
1.2.3	What are the disadvantages of SpeechBuilder? . . . . .	20
1.2.4	How does SpeechBuilder work? . . . . .	21
1.3	Data Acquisition . . . . .	22
1.4	Overview . . . . .	23
<b>2</b>	<b>Background</b>	<b>24</b>
2.1	Generic Dialogue Server . . . . .	24
2.2	Dialogue Control . . . . .	26
2.2.1	Pre-Retrieval . . . . .	26
2.2.2	Retrieval . . . . .	27
2.2.3	Filtering . . . . .	27
2.2.4	Response Generation . . . . .	28
2.3	Discourse . . . . .	28
2.4	Frame Relay . . . . .	29

<b>3</b>	<b>Approach</b>	<b>32</b>
3.1	Dialogue Control configuration . . . . .	32
3.2	Helping Developers use Dialogue Control . . . . .	35
3.2.1	Evolution of DCTL interface in SpeechBuilder . . . . .	36
3.2.2	Incorporated DCTL file . . . . .	41
3.2.3	Voice Configuration file . . . . .	42
3.3	Comparison of VCFG and XML as formats for domain specification .	43
3.4	Frame Relay . . . . .	45
<b>4</b>	<b>Results</b>	<b>48</b>
4.1	Configuration of Dialogue . . . . .	48
4.1.1	SpeechBuilder . . . . .	48
4.1.2	Voice Configuration . . . . .	52
4.2	Configuration of Discourse . . . . .	52
4.2.1	SpeechBuilder . . . . .	53
4.2.2	Voice Configuration . . . . .	54
<b>5</b>	<b>Applications</b>	<b>56</b>
5.1	Echo . . . . .	56
5.2	Hotels . . . . .	57
5.3	Mars . . . . .	60
5.3.1	Database Script . . . . .	60
5.3.2	Scores . . . . .	62
5.3.3	Standings . . . . .	64
5.3.4	Inheritance . . . . .	65
5.3.5	Evolution of Mars Server . . . . .	67
<b>6</b>	<b>Conclusion</b>	<b>68</b>
<b>A</b>	<b>Mars Domain</b>	<b>74</b>
<b>B</b>	<b>List of files generated by SpeechBuilder</b>	<b>78</b>



# List of Figures

1.1	Galaxy Architecture . . . . .	12
1.2	Example discourse frame, where the phone number for “Jane” has been requested, and the phone number for John Smith was most recently given. . . . .	16
1.3	Interface for specifying Actions in SpeechBuilder . . . . .	19
2.1	Example of domain built in SpeechBuilder without configured dialogue	27
2.2	Example of domain built in SpeechBuilder with configured dialogue .	27
2.3	Example of domain using dialogue with filtering . . . . .	28
2.4	Example of domain using discourse without masking . . . . .	29
2.5	Example of domain using discourse with masking . . . . .	29
2.6	Frame Relay . . . . .	30
3.1	Sample frame from generic dialogue server that shows which attributes have values . . . . .	34
3.2	Dialogue Control v.0 . . . . .	36
3.3	Dialogue Control v.1 . . . . .	38
3.4	Dialogue Control v.2 . . . . .	39
3.5	Dialogue Control v.3 . . . . .	40
3.6	Dialogue Control v.3 . . . . .	40
3.7	Dialogue section of Voice Configuration file . . . . .	42
3.8	Sample transaction between Hub and Frame Relay, with access to external data server . . . . .	46

4.1	Dialogue Control v.2 . . . . .	49
4.2	Dialogue section of Voice Configuration file . . . . .	51
4.3	Current interface for discourse configuration in SpeechBuilder . . . . .	53
4.4	Discourse specification in Voice Configuration file . . . . .	54
4.5	Example of file which specifies inheritance . . . . .	55
5.1	Excerpt of dialogue control file which specifies “echo” as the data server	57
5.2	Sample interaction from Hotel information domain . . . . .	58
5.3	Sample interaction from Sports information domain . . . . .	60
5.4	Sample <code>db_result</code> response from the Mars server. The value shown is a “game frame,” representing the baseball game between the Cleveland Indians and Minnesota Twins played April 8, 2002 . . . . .	63

# Chapter 1

## Introduction

Recently, my wife and I were driving the distance from Boston to Washington, D.C. During the drive, we listened to several talk shows, ranging in topics from pets to politics, but nothing about sports. To find out the score of my favorite hockey team's game, I dialed in to the Mars sports score system and asked the question, "Did the Canucks win their game tonight?" The response, "The Red Wings beat the Canucks, 6-4," came as unwelcome<sup>1</sup> news, but at least the information was only a phone call away.

The Mars service I called is known as a **conversational system** [1]. A conversational system is an interface with which a person interacts by way of natural, "conversational" speech. Such systems are growing in popularity, but the technology required to support and create them is very complex, and is still developing.

The goal of this project is to make domains such as Mars easier for novice and expert developers to build and use. This includes making all of the necessary functionality available through the existing development tool, called SpeechBuilder, and helping developers understand how to make use of that functionality. This chapter will introduce the key concepts of this project, including the framework of conversational systems, the tool available for building conversational systems (SpeechBuilder), and the components which have not previously been configurable through SpeechBuilder.

---

<sup>1</sup>because it signified the end of the Canucks' 2002 playoff run. In fact, my wife wanted to know the score as well, because I had promised not to shave until the Canucks were eliminated.

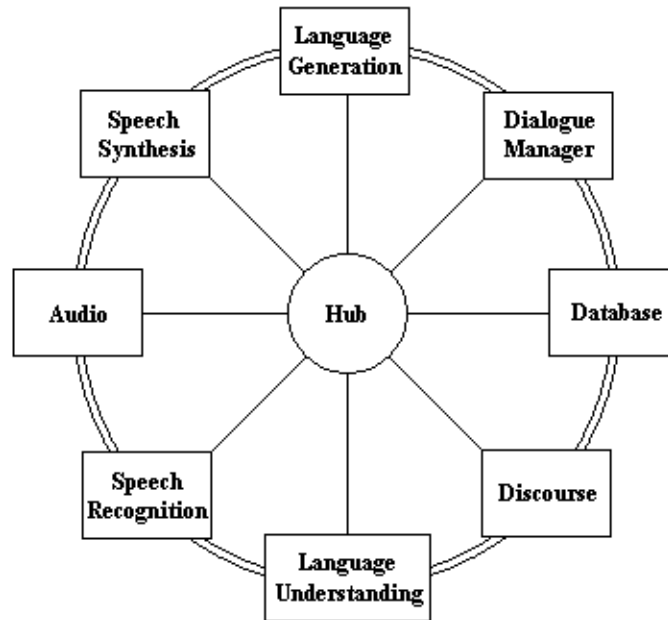


Figure 1.1: Galaxy Architecture

## 1.1 Galaxy

The MIT Spoken Language Systems group has developed an architecture for building conversational systems, called Galaxy [2]. The Galaxy architecture, shown in Figure 1.1, brings together the necessary components for a conversational system to fully function. The pieces of this system, however, are so complex that a developer needs to be expert in some or all of them in order to put a full system together.

There are nine different components of a Galaxy system, as used at MIT, which are as follows: the audio capture module, speech recognition, language understanding, context resolution, the database, dialogue management, language generation, speech synthesis, and a hub. These components are arranged in a hub-and-spokes model; the eight servers interact with each other primarily through the hub.

A typical exchange between a user and a Galaxy system might flow as follows:

**User:** (Spoken) “Did the Canucks win their game tonight?”

**Audio:** Converts spoken input from user into digital data.

**Speech Recognizer:** Transforms input data into 10 most likely sentences.

**Language Understanding:** Parses sentence into attributes and values, such as `team=Canucks`, `clause=request_score`, and `date=tonight`.

**Discourse:** Does nothing in this case, but if the user had previously asked about that day’s score for a different team, then said, “How about the Canucks,” it would have kept `date=tonight` and `clause=request_score` from the previous statement.

**Database:** Retrieves the score from the data provider. In this case, it would retrieve `winningteam=Red_Wings`, `losingteam=Canucks`, `winningscore=6`, `losingscore=4`, `decision=win`.

**Dialogue Manager:** Determines how to respond to the user, based on the information from the database server. The appropriate template for a response from the dialogue manager in this case is: “The `:winningteam` beat the `:losingteam` `:winningscore` `:losingscore`”

**Language Generation:** Translates the template sentence into something understandable by the user. “The Red Wings beat the Canucks 6 4.”

**Speech Synthesis:** Creates the spoken output audio.

**Audio:** The output sentence is spoken to the user.

Traditionally, a developer had to have a detailed understanding of each of these servers, because manual configuration was required for each one. This is a time-intensive and tedious process, because the configuration is the same for many domains, but each server must be configured correctly for a system to function.

To help make system development simpler, the process for configuring these servers can be made more automatic. By automating as much of the development process as possible, novice developers can access the components of the Galaxy architecture without understanding unnecessary details. At the same time, expert developers can focus on the unique aspects of their specific applications, without facing the tedium

of repeating similar work from application to application.

The tool available for developing conversational systems, SpeechBuilder, currently configures all of the servers with the exception of discourse and dialogue. The configuration of dialogue, in particular, has been difficult to automate, because in many cases it requires writing and compiling functions which are specific to a domain. With this in mind, our goal has been to add dialogue and discourse configuration to SpeechBuilder.

### 1.1.1 Dialogue

In common terminology, a *dialogue* is a conversation between two people. In conversational systems, and for the purposes of this thesis, *dialogue* will refer to the conversational element between a user and the system, which includes anything which is spoken by the user and that which is spoken back to the user by the system. Among the most important aspects of dialogue are requests for information (made by the user or the system), and the responses to those requests.

The dialogue in a conversational system is controlled by a *dialogue manager* [5]. The dialogue manager is the component of the system which determines when the user has not provided sufficient information, and which identifies the best way to return information to the user when appropriate information is available.

In a conversational system, the dialogue control typically has four key components or phases, which are as follows: Pre-Retrieval, Data Retrieval, Data Filtering, and Response Generation. The implementation of these phases is discussed in further detail in Section 2.2.

Although SpeechBuilder can effectively build systems which answer direct questions, it is currently weak in the management of dialogue. For example, a domain which is built in SpeechBuilder can not prompt the user for missing information. If a user called an airline reservations domain and asked for “a flight,” the system would not know how to ask for the departure or arrival cities, because none of the compo-

nents which are configurable through SpeechBuilder support that functionality. The technology for a more sophisticated dialogue has been developed for use in other domains using the Galaxy architecture, but it is not accessible through SpeechBuilder.

Although it can make dialogue configuration easier, SpeechBuilder will probably never completely replace the need for manual coding on the part of the developer. The unique aspects of the domain, however, are exactly what the developer should be able to focus on, as opposed to those parts of the system which can easily be generated automatically.

### 1.1.2 Discourse

A conversational system which uses *discourse* maintains a state based on previous utterances and system calls [7]. This can also be seen as the “memory” of the system, concerning what has been learned for or from the user. For example, if a user asks about flights leaving Boston tomorrow, then asks about Manchester, the system will remember that the user was asking about flights leaving tomorrow.

In many ways, the dialogue and the discourse of the system work very closely together as the processing center<sup>2</sup> of a Galaxy system. Indeed, it plays a large role in making the system seem more like a real person. Domains built in SpeechBuilder typically have a discourse server which remembers all of what a user says, and carries it forward as long as the session exists.

In normal human conversation, however, there are times when a person “forgets” or ignores certain things which are no longer relevant. For example, a visitor in a workplace asks the receptionist for John’s extension number. Because there is only one John, the receptionist knows to give the visitor John Smith’s extension. If the visitor then asks for Jane’s extension, the receptionist does not assume the last name is Smith. It is assumed the visitor is not necessarily referring to Jane Smith, so the last name is discarded from memory, and not automatically applied to the new query.

---

<sup>2</sup>Somewhat like the brain. The Dialogue “reasons” while the Discourse “remembers.”

```

{c discourse.discourse
  :key_value "clause: request_property name: Jane property: phone"
  :parse_frame {c request_property
    :property "phone"
    :first_name "Jane" }
  :history_frame {c history
    :history_id 0
    :input_frame {c request_property
      :property "phone"
      :first_name "John"
      :last_name "Smith"
      :name "John Smith" } } }

```

Figure 1.2: Example discourse frame, where the phone number for “Jane” has been requested, and the phone number for John Smith was most recently given.

In contrast, a computer which makes use of context would maintain this information, and look for Jane Smith. An example of a discourse frame that represents this situation is found in Figure 1.2.

Why does the receptionist know not to look for Jane Smith, and to instead look for any Jane? The reason is common sense; this is how people commonly communicate with each other. A computer, of course, is not a real person and therefore must have this type of common sense information explicitly stated. Although natural language servers have supported this functionality, it was not available through development interfaces previous to this project.

## 1.2 SpeechBuilder

A utility called SpeechBuilder was created to help developers build conversational systems [3]. It is a web-based graphical interface with which conversational systems can be designed, compiled, and run. SpeechBuilder is a utility which has all the configurable components of a conversational system in one place. This helps developers quickly configure a domain, and it also functions as a structured environment for a new developer who wishes to learn about domain creation.



### 1.2.1 What is SpeechBuilder?

SpeechBuilder is a set of different interfaces, each one of which modifies a different part of a conversational domain. The interfaces are as follows: Attributes and Actions, Responses, Database Management, Vocabulary, Dialogue Control, and Context Resolution. Dialogue Control and Context Resolution are new modules, and will be discussed in more detail later. SpeechBuilder also includes a set of scripts which are useful to get a domain running.

**Attributes** are concepts that the system will need to recognize. Mars, for example, needs to recognize team names and cities. It also needs to recognize special keywords, known as **properties**, such as “win,” “won,” “standings,” “lose,” etc.

**Actions** are different requests or functions the system can perform. In a movie domain, there might be such actions as *request\_titles* or *request\_theatres*. Each action has an associated list of phrases that the system will recognize. *request\_titles* could have “What movies are playing?” and “What can I see at the Fenway Cinema?” listed as associated phrases. When the system receives an utterance from the user that matches one of these sentences, it will invoke *request\_titles*.

**Responses** are the sentences which are spoken back to the user when the system needs to either prompt for or reply with information. These can be either full sentences or sentence fragments. For example, the system could be configured to use the following response after performing a *request\_titles* action: “The following movies are playing at `:theater` `:items`” The system would insert the name of the theater (`:theater`), then list the movies returned from the data source (`:items`).

The developer uses **Database Management** to upload tables of data into the domain. This is most useful when the data required for the domain is contained in a static table. When the domain is compiled, this data is stored in a relational database which is then accessed by the system whenever a database call is performed. The data is also stored by SpeechBuilder, so that the developer does not have to upload the same information more than once.

The **Vocabulary** consists of all the words which are attributes or actions. Initially, the system generates a default pronunciation for each word, and this interface allows the developer to modify the pronunciation of these words.

Once the developer has configured all of the necessary components for the domain, SpeechBuilder also provides a mechanism for activating the system, which means that users could begin to interact with it. This involves *compiling* the domain, and then *running* it.

When a domain is *compiled* in SpeechBuilder, the system uses the information that has been specified by the developer, and it generates all of the necessary configuration files for each server. The list of files which are generated is shown in Appendix B. These files include text files, with lists, word pairs, etc.; representations of finite state transducers (FSTs); and executable servers, among other file types. When the compile is executed in SpeechBuilder, this file creation process is completely automatic, and requires no additional input from the developer.

When a domain is *run* in SpeechBuilder, all of the necessary servers are executed automatically. This includes using the correct command-line parameters so that the servers will recognize the words which are specific to the domain, perform the correct actions, etc.

These domains do not have to be compiled or run through the SpeechBuilder interface; the SpeechBuilder compiler invokes scripts which already exist, and the same is true for running the domain. That these functions are available in SpeechBuilder means that developers can complete the process of domain creation and execution through a single interface.

### **1.2.2 Why is SpeechBuilder useful to developers?**

SpeechBuilder maintains all of the information pertaining to a domain in a single *domain file*, formatted in XML. Whenever a developer lists a set of attributes or actions, etc., these are saved to the domain file. When a developer loads a domain in

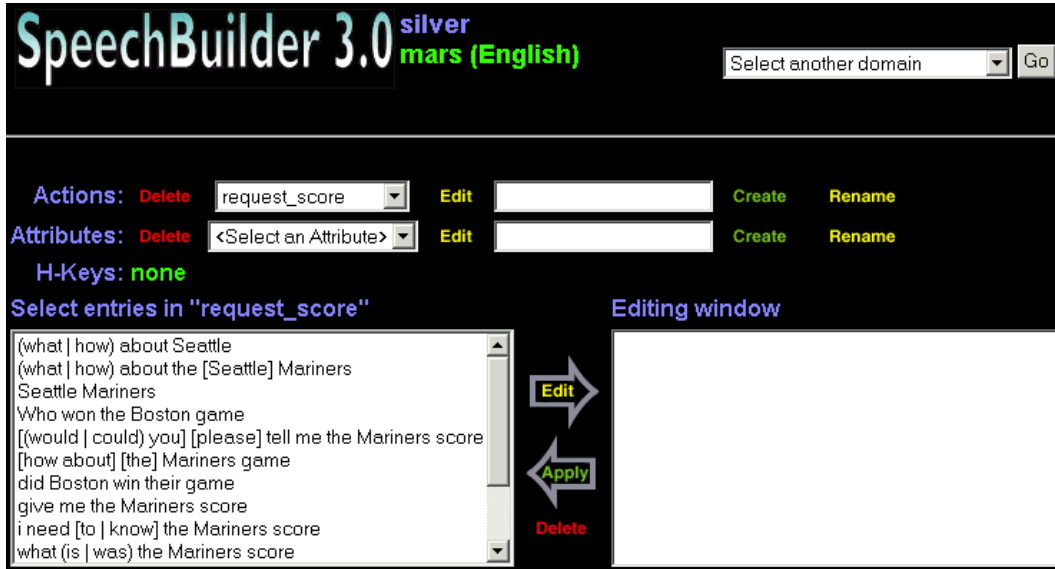


Figure 1.3: Interface for specifying Actions in SpeechBuilder

order to make changes or additions, the whole domain is loaded from the same file. This arrangement provides for great portability of domains, because anyone who has access to either SpeechBuilder or the scripts required for compiling a domain file can take any such file, compile it, and have the entire domain.

SpeechBuilder is also designed in such a way that a person who wishes to develop a domain needs very little programming, domain building, or natural language experience. The concepts are presented in a way that is so simple that it is very easy to quickly understand domain creation, or make changes to a domain with little or no formal training. The interface for adding Actions is shown in Figure 1.3.

One of the great advantages of using a utility such as SpeechBuilder for novices and experts alike is the time saved in debugging files during domain creation. For example, it was explained above that all the words which are attributes, actions, etc. are listed with pronunciations. The nature of a conversational system leads to this situation, because there are separate servers for many different tasks. For instance, in the Galaxy architecture, there is a server which handles recognition, another server which handles pronunciation, and another server that understands

language, etc. Each of these servers has its own configuration files which refer to common attributes, actions, and words. If these files become inconsistent with each other, then problems will begin to arise.

This type of scattered configuration is fairly prevalent among conversational systems, so it is very convenient to specify something once and have it correctly configured in all of the necessary places. In an alternate scenario, without an automatic configuration utility, a developer might remember to specify something in four out of five required files, then spend an exorbitant amount of time debugging and searching for why the domain does not recognize something obvious. This is difficult for expert developers, and practically impossible for novices.

As a result, SpeechBuilder makes modifying an existing domain much easier for developers. Changing the spelling of a misspelled word or adding a new phrase is no longer a long search-and-replace effort among a myriad of files; the developer simply finds the word in SpeechBuilder and makes the change. The compiler handles everything else.

### **1.2.3 What are the disadvantages of SpeechBuilder?**

SpeechBuilder provides a simple interface for the specification of all components of a conversational system in the Galaxy architecture, aside from discourse and dialogue. In fact, SpeechBuilder was designed to help novice developers get started more quickly, and to help expert developers avoid the more tedious parts of domain creation. For expert developers, however, SpeechBuilder does not always afford enough flexibility to be able to create a desired domain. It serves very well as a starting point, because compiling a domain creates all of the files necessary for a domain to function. At this point, however, it becomes less useful for an expert developer, because in many cases, the developer will wish to manually modify the automatically generated files. As soon as this occurs, part of the usefulness of SpeechBuilder is lost for that domain.

As mentioned earlier, SpeechBuilder maintains all of the information relevant to a

domain in a single file. The current SpeechBuilder assumption is that all domain development takes place via its own interface, and any time a developer makes changes to his or her domain in SpeechBuilder, the changes are saved to this file. When a SpeechBuilder compile is executed, any necessary files are automatically generated again, regardless of whether they already exist. Therefore, if the developer has made manual modifications to any of the files which were previously generated by SpeechBuilder, these files will be overwritten by the compiler unless they have been write-protected because their changes will not have been incorporated into the single domain file. SpeechBuilder interacts exclusively with this domain file, and will never learn of the changes that were made manually. The result is that once a developer has modified files outside of SpeechBuilder, it is very difficult to return to a state where SpeechBuilder is useful because the domain file no longer represents the domain as the developer has built it.

Fortunately, none of these disadvantages appear to impact novice developers very greatly. Furthermore, developers who are truly experts can likely fix any potential issues with some minimal effort. It appears that the developers with the most to lose are those who are somewhat experienced, but do not have a solid understanding of all the system configuration files.

#### **1.2.4 How does SpeechBuilder work?**

As explained earlier, the primary result of a SpeechBuilder compile is that all of the necessary files for a domain system configuration are automatically generated. The list of files is shown in Appendix B. Compiling a domain is performed by executing the script, `process_xml`. It, in turn, calls several other scripts, including `xml2n1`, which builds the natural language framework for the system based on what is in the domain file.

Some servers in the conversational system framework, such as the speech recognizer, require configuration files which are specific to the domain. For example, the

speech recognizer may need to be able to recognize words that are not in a standard vocabulary. In cases where a server will need domain-specific information, the compiler generates configuration files based on the information stored in the domain file.

In other cases, all that is required for a server to function correctly is a file which is not changed in any way based on the specifications of the developer. This depends to an extent on what has been specified by the developer, but the dialogue manager, for example, only needs to know the name of the domain to function correctly.

As a matter of convenience, a SpeechBuilder compile also produces the scripts necessary for running a domain either using local machine audio or the development telephony server. These scripts perform such tasks as starting each of the nine servers required for a conversational system, initiating the communication between these servers, and passing the correct arguments to each server. This keeps the developer from having to individually execute the servers individually or having to remember all of the necessary command-line arguments. Additionally, if a developer wishes to debug his or her domain, these scripts can be invoked in such a way that all the output messages are displayed.

In summary, compiling a domain produces everything a domain needs to run. If something has not been specified by the developer, then the compiler works around the gap by generating files that will keep the domain from crashing.

### **1.3 Data Acquisition**

Domain developers can use one of two methods to retrieve data. First, the developer can create a relational table which the system accesses with via standard SQL queries. If this does not allow enough flexibility for the developer, or if the content is dynamic, he or she can instead specify a script which will retrieve the data and produce responses.

In the case of an external database, the developer can either specify the informa-

tion contained in the database through SpeechBuilder itself, or the database can be configured separately. Data can be updated dynamically straight to the database, but not through SpeechBuilder.

With a CGI script as the data provider, much more flexibility is offered to the developer, with much less structure. This script could retrieve the data by any means necessary; it could access a database or even get the data from a website. Essentially, the developer is left to do almost everything through the script, and much of the benefit of using SpeechBuilder as a configuration tool is lost.

## 1.4 Overview

SpeechBuilder meets many needs of a domain developer, but it does not provide an interface for modifying dialogue or discourse, as mentioned above. Therefore, the goal is to add these two components to SpeechBuilder, so that the more complex domains can be built by less experienced developers, and expert developers can spend less time on the tasks which are repetitive among domains.

The rest of this thesis will explain how the goals outlined above are achieved. Chapter 2 is a description of the technologies which are available for conversational systems built with the Galaxy architecture. Chapter 3 gives an explanation of the different methods we tried to make this technology available to developers. In Chapter 4, the results of these efforts are given, and Chapter 5 shows some of the domains that were built using the newly developed tools.

# Chapter 2

## Background

Up to now, SpeechBuilder has made virtually no use of the dialogue and discourse capabilities which can be built into conversational systems. The primary goals for this project were to add dialogue and discourse configuration to SpeechBuilder, and to make these understandable and configurable for the developer.

What follows is a description of the various technologies that are available for use within the Galaxy framework, that have not been previously integrated into SpeechBuilder. These include the generic dialogue server, dialogue control, the discourse server, and a new server called the frame relay.

### 2.1 Generic Dialogue Server

Before the development of the **generic dialogue server**, every domain built in the Galaxy architecture had to have a domain-specific dialogue manager. This meant that a developer had to write, create functions for, and compile a new dialogue server every time he or she made a new domain.

Of all the components, dialogue management is the one which has most resisted the move to portability, because everything about it is so domain-specific. A flight reservation domain, for example, could have a need to “find a return flight” or “speak only the itinerary price.” Such operations as these clearly do not apply to other domains such as a sports scores domain. As systems have been built, however, certain



dialogue functionalities have emerged, such as result filtering and database querying which are useful across many domains. These types of functionalities have been brought together in one single *generic* dialogue server [4].

In fact, the SpeechBuilder effort of making domain development more feasible for novice developers is what initially prompted the creation of the generic dialogue server. The introduction of this server to the Galaxy architecture is what has truly made it possible to reasonably configure a dialogue server in SpeechBuilder.

Functions which are more generic, and are useful across the spectrum of domains, are now compiled into this generic dialogue server. An example of this would be as follows: under the domain-specific framework, a developer might have written the functions `rank_teams_by_division` and `rank_teams_by_league` in order to return the division and wild-card standings in the Mars domain. Using the generic dialogue server, however, it is necessary only to use more generic pre-existing functions like `filter_db_tlist` and `rank_by_key`, and specify the appropriate parameters. For example, if the system filters the results using the division name as the parameter, and ranks them by the number of games back, this is effectively the same as `rank_teams_by_division`. The latter case does not require any coding on the part of the developer; he or she needs only to know which generic functions are available, and specify the correct arguments in the dialogue manager.

The generic dialogue server, of course, does not take away from the developer's ability to add custom functions where they are desired. It certainly has not removed all need of such functions, but it does open the door for less experienced developers to easily create more sophisticated systems using more complex dialogue, because it requires less detailed knowledge of the dialogue management technology. Domains built with this more complex dialogue are capable of filtering data, prompting the user for missing information, etc., while domains built in SpeechBuilder without dialogue control do not have this capability.

Originally, SpeechBuilder automatically compiled a domain-specific dialogue server

for each domain. This server did not make use of much dialogue functionality. Instead, it simply served the purposes of executing SQL queries to a database, and returning the correct response to the user depending on the number of results from the database. It did not do any constraint checking before performing requests, and it did not filter any results. With the addition of the generic dialogue server, developers can now make use of those types of functionality through SpeechBuilder.

The generic dialogue server also represents the graduation of dialogue management into the group of servers which are not domain-dependent. Given this, SpeechBuilder no longer needs to compile a new dialogue server for each domain, but can instead create the files which are necessary for the dialogue server to process information correctly. This also increases the reliability of the overall system, because the dialogue server can now also become a well-tested and proven component.

## **2.2 Dialogue Control**

The introduction of dialogue control to SpeechBuilder enables a much more sophisticated and natural conversation between the user and the system. The dialogue manager has four primary components, which are listed and explained here:

### **2.2.1 Pre-Retrieval**

In many cases, before a query can be made to a data source, there is a certain set of input information which must be provided. For example, in a Hotel information domain, to ask about available hotels could require the name of the city where the database should search for availability. This would be necessary if the website where the information is obtained has this requirement. Therefore, if the user has not specified a city, the dialogue manager should be aware of this requirement before attempting to access the data source.

If the dialogue manager determines that some input is missing, it prompts the user for that information. This is a clear example of how a more sophisticated dialogue

**User:** "I need a hotel."  
**System:** [queries database] "Please be more specific."

Figure 2.1: Example of domain built in SpeechBuilder without configured dialogue

**User:** "I need a hotel."  
**System:** "Please specify a city."  
**User:** "Boston"  
**System:** [queries database] "I have 15 entries matching your request..."

Figure 2.2: Example of domain built in SpeechBuilder with configured dialogue

manager results in a more natural conversation between the user and the system. Figure 2.1 is an example of a domain built in SpeechBuilder, in which the dialogue management was not configured, and Figure 2.2 is an example using the new dialogue management.

Without this dialogue management, the system leaves the user helpless to determine why there are no hotels. Adding it makes the system considerably more intelligent, and, more importantly, helps the user achieve the desired end goal: a hotel in Boston.

### 2.2.2 Retrieval

The dialogue manager is the component of Galaxy that actually instructs the system to query a database. Based on the state of the system, it can query any one of a number of data sources, and it can send any set of attributes to the database.

### 2.2.3 Filtering

This portion of the dialogue management organizes the results from the database into a format which is useful to the user. This can include omitting results that do not match the user's criteria, putting them in a ranked order, etc. Again, the result is that the response from the system is much more intelligent, and gives a higher priority to the information that the user wants most. The example given in Figure 2.3 follows from the example given in Figure 2.2, but makes use of filtering.

<p><b>System:</b> “I have 15 entries matching your request. Please specify a brand.”</p> <p><b>User:</b> “How about a Sheraton.”</p> <p><b>System:</b> “There are 7 hotels matching your request. Please be more specific.”</p> <p><b>User:</b> “Give me the cheapest.”</p> <p><b>System:</b> “This is the cheapest hotel matching your request: Airport Sheraton at 99 dollars per night...”</p>
---

Figure 2.3: Example of domain using dialogue with filtering

The user could have conceivably listened to all of the results and all the associated information returned from the database, but by using filtering and ranking the best entries are spoken first. This enables the user to more easily have his or her question answered and understood, which is the goal.

## 2.2.4 Response Generation

A response to the user involves the construction of a reply frame with results from the database, other relevant information, follow-up questions, or any combination of these three components. As shown in Figure 2.3, the system found 15 results from the database, but it did not speak them all back. Instead, it constructed a reply to give the user information (there are 15 results), then ask for the brand (follow-up question). At the end of the example, when only one result remained after filtering, it was spoken back to the user.

Had all 15 entries initially been read to the user, it would likely have been an overwhelming and useless response. Instead, the system made the user aware of the number of entries and gave the instruction to be more specific. On a web page, it might be useful to see all 15 entries at once, but over the telephone a user is much more likely to want only the best one or two entries.

## 2.3 Discourse

Previous versions of SpeechBuilder also had no way of specifying context inheritance and masking outside of having it built into a CGI script, although it is a relatively

**User:** “Did the Boston Red Sox win their game last night?”  
**System:** “The Red Sox beat the Blue Jays 5-4.”  
**User:** “How about the Yankees?”  
**System:** “I do not recognize the Boston Yankees. Is it possible you meant to say Boston Red Sox, Boston Celtics, Boston Bruins, or New York Yankees?”

Figure 2.4: Example of domain using discourse without masking

**User:** “Did the Boston Red Sox win their game last night?”  
**System:** “The Red Sox beat the Blue Jays 5-4.”  
**User:** “How about the Yankees?”  
**System:** “The Yankees lost to the Devil Rays 8-0.”

Figure 2.5: Example of domain using discourse with masking

simple concept. In Figure 2.4 is an example of system that does not use discourse with masking, and the same system with masking is shown in Figure 2.5.

Because the default behavior of conversational systems using this dialogue is to inherit any information from the previous frame, it is the developers responsibility to specify when not to inherit, which is known as masking. The above example shows the importance of masking; without it certain domains would be virtually unusable.

## 2.4 Frame Relay

In any data-based domain, the data provider is extremely important, because without it the domain is useless. Originally, a developer could meet these needs by either putting the data into a relational database, or by using a CGI script. Now, there is a third, more flexible, mechanism known as the **Frame Relay**.

The frame relay does exactly what its name indicates. It accepts a frame from one server, and it passes it to another. It could be used for many different functions within Galaxy, but to this point it has primarily been used as a substitute for the database server.

Another benefit of using the frame relay instead of the database server is that it is possible to add new data sources without having to restart the frame relay.

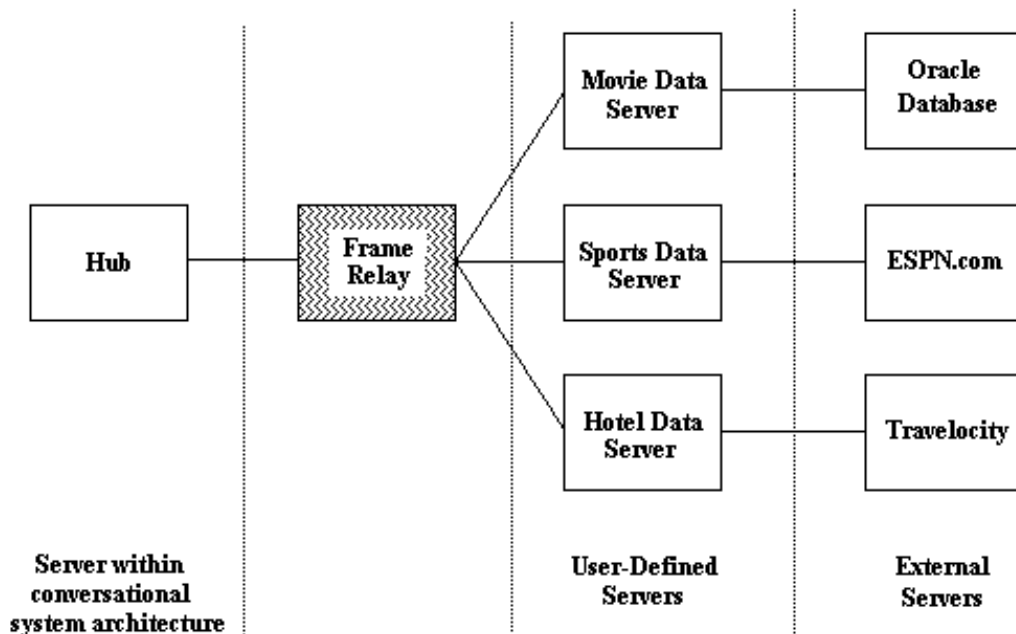


Figure 2.6: Frame Relay

This is because it has no assumptions about which sources it will have to access. The traditional database server must be restarted every time a new source is added, because the data sources must all be specified in a text file which is loaded at runtime. This means that the frame relay does not have to be restarted, even when a SpeechBuilder domain is recompiled.

The addition of the frame relay provides a new level of abstraction to the Galaxy architecture. Writing a script that can act as a database server and interface with the hub would be a daunting task for a novice, because there are many ways that the hub might expect to be able to interact with the database server. Moreover, a developer does not have to re-specify all the necessary database functions each time he or she creates a new domain.

Take, for example, a domain in which the developer wishes to provide information about rental car availability. He or she might write a server that waits for a city name and brand from the frame relay, then extracts the relevant data from a website. The

data is then returned in a frame to the frame relay, and the frame relay forwards the data to the appropriate server from that point.

As shown, the developer can write a simple data broker, instead of having to create a full database server. One only needs to know how to interface with the frame relay, and can focus on the details of providing data.

Every server that connects to the frame relay, such as the rental car server given above, has a name. Any number of servers can therefore be connected to the same frame relay, as long as they all have different names. This property makes it easier for a developer to create a domain that can access several databases. In fact, adding another database can be specified in the dialogue manager with a single instruction.

# Chapter 3

## Approach

We developed several different SpeechBuilder interfaces for dialogue control. The primary goals were to make the dialogue control interface as flexible as possible, and to make it as understandable as possible. Therefore, in the effort to bring this functionality to novice developers, great lengths have been taken to present dialogue management in a clear, simple, and complete manner.

This chapter is a discussion of the different attempts to achieve these goals, what worked well, and what was particularly challenging. Along with several different SpeechBuilder interfaces which were developed, a new format for domain specification resulted. There is also an explanation about the frame relay, and the role it plays in helping developers to more easily develop domains with dynamic data sources.

### 3.1 Dialogue Control configuration

Dialogue is an extremely powerful component of a conversational system. In order to create domains which take full advantage of the available technology, this functionality must be accessible to developers. Dialogue is also fundamentally different from the other components that developers may configure in SpeechBuilder. The fundamental difference lies in the fact that a dialogue manager is essentially a state machine. It uses the state which is maintained in the discourse to decide when to invoke actions. Then, depending on which information is missing or present, it sends a message back



to the user.

For a developer to configure the dialogue manager requires knowledge of a few things. First, one must be aware of the different keys and values which are created by the system or received from the data server. See Figure 3.1. In this example, `:name` is information which was submitted by the user, and `extension` was obtained as part of a result from the database. The developer must also know the types of these keys, whether they be integers, strings, etc.

Next, it is necessary to have an understanding of the functions that are available for use in the generic dialogue server. Some example functions are `execute_db_query`, which makes a database query, `query_for_constraints`, which solicits information from the user when necessary, `speak_no_data`, which informs the user that no data has matched his or her request, and `filter_db_tlist`, which filters results from the database based on the value of a particular attribute. Additionally, some functions have parameters which must be specified in order for them to work correctly. `filter_db_tlist`, for example, needs to have a `filter_key` specified so that it will filter for the right attributes. Previously, it was necessary for a user to create such functions, but with the new generic server these are provided for the developer. An expert developer can produce custom functions if they are necessary.

Having an understanding of the keys and functions, a developer can then proceed to specify the dialogue state machine. Configuring a dialogue manager in a conversational system is clearly very different from configuring other types of information, such as attributes and actions. Attributes only need to be listed, and the system can scan for them in utterances from the user. This is relatively simple to present to a developer, whereas learning all of the different parts of dialogue control is very complicated.

We have also developed a new domain specification format, called Voice Configuration (VCFG), which makes the assumption that a domain developer would be comfortable with less of a graphical interface, but that they still need assistance

```

{c db_query
  :domain "SpeechBuilder"
  :para_lang "sql"
  :db_request {c eform
    :num_found 1
    :clause "request_property"
    :name "John Smith"
    :property "extension"
    :domain "PhoneInfo"
    :first_name "John"
    :last_name "Smith"
    :extension "x5084"
    :office "645"
    :init 1
    :selected_input_string "What is John's Extension?"
    :para_lang "sql" }
  :session_id "18.27.0.95:22213:L0:0"
  :paraphrase_string "eform"
  :db_result {c info
    :nfound 1
    :values ( {q employee
      :first_name "John"
      :last_name "Smith"
      :extension "x5084"
      :office "645" } ) }
  :hub_opaque_data {c admin_info
    :tidx 15
    :session_id "18.27.0.95:22213:L0:0"
    :serve_tidx 1 } }

```

Figure 3.1: Sample frame from generic dialogue server that shows which attributes have values

with the natural language aspects of domain configuration. It consists of a single file which contains all the components of a domain which can be specified by the developer. Compiling a VCFG file has the same result as compiling a domain in SpeechBuilder.

What follows is a description of the different attempts at dialogue configuration interfaces.

## 3.2 Helping Developers use Dialogue Control

In conversational systems developed at SLS, the dialogue control is specified in a single file. This file is an ordered list of instructions, each of which consists of a predicate and a resultant function. The generic dialogue server accesses this file at run-time, and executes each function in order when predicates are satisfied.

An example dialogue configuration file might have a list of instructions that first, verifies that a user has named a city; second, retrieves the list of hotels in that city; third, filters out the hotels that do not match a user-specified brand; and fourth, speaks the list back to the user.

It is extremely likely that a new developer would not know how to use this configuration file for dialogue control, because of its structure. If the developer understands the purpose and function of dialogue, however, modifying this file manually is no more difficult than it would be to configure dialogue with a graphical interface. In a sense, this is the opposite of the problem that SpeechBuilder is optimized to solve.

In general, SpeechBuilder takes concepts that the developer already understands and presents them in a way that makes them easy to configure. This is important because the number of files required for a single domain is large, and the same information must be represented in multiple locations in order for the system to work correctly. Therefore, an automated utility for creating them is extremely useful for novices and experts alike.

Configuration of dialogue control, however, requires only the modification of a

Use this page by typing the DCTL file into the text area exactly how you would expect to do it in a text editor.

```
!!:init --> initialize_server
:verifying --> process_verifying
--> check_confidence_scores
:nth --> set_up_nth
!!:nth --> set_up_selected
:clause request_score & !:team_name & !:city | :reply_frame need_team -->
query_for_constraint
:clause request_standings & !:division | :reply_frame need_division -->
query_for_constraint
!date & !:nth & :num_found <0 --> maybe_add_week_to_date
!:selected & num_found <0 | :query_language espn | :destination mars -->
evaluate_db_query
:num_found >0 & !:nth & !:selected | :filter_key team_name | :db_key teams
| :match_condition strstr --> filter_db_tlist
:num_found >0 & !:nth & !:selected | :filter_key division | :match_condition
strstr | --> filter_db_tlist
:filtered | :rank_key games_back --> rank_by_key
:clause take_hotel --> maybe_add_system_initiative
:property & :num_found =0 --> speak_no_data
:clause %request & !:property & !:boolproperty --> speak_need_property
```

Figure 3.2: Dialogue Control v.0

single file, as explained earlier. The difficulty for the developer is in understanding the function of dialogue, and it is therefore not only important that the dialogue be easy to configure in SpeechBuilder, but that it be easy to understand.

### 3.2.1 Evolution of DCTL interface in SpeechBuilder

The dialogue control module of SpeechBuilder has undergone considerable evolution since its initial creation, and this evolution is evidence of the difficulty in effectively presenting dialogue functionality to the developer. The interfaces described in this section reflect the two primary goals, which were to make dialogue configuration as flexible as possible in SpeechBuilder, but also to make learning about dialogue control less overwhelming for a novice developer.

The first interface was simply a text box in SpeechBuilder that stored the text to the domain in whatever form it appeared in the box. This approach was useful because it offered maximum flexibility to the developer. In contrast, however, it gave no indication as to what was expected from the developer. It was an interface which was essentially equivalent to editing the dialogue control file in a text editor.

The next interface consisted of a similar buffer on the web, but it included a mechanism for adding commands in a structured fashion. This is shown in Figure 3.3. That is to say that the user could either edit the dialogue control in the text buffer, or he or she could add lines by specifying arguments and boolean operators in the interface below.

This approach had several new benefits. First, it allowed the user to edit any part of the file as they would otherwise have done in a text editor, but the new part of the interface helped the user gain a more intuitive understanding of dialogue management and the underlying state machine by seeing how the values of certain attributes lead to the execution of functions. Also, by using this automated section, the developer could avoid unnecessary syntactical mistakes in the configuration. This design still provided a complete implementation of dialogue control; there was nothing a developer could not configure.

Version 2 was completely based on a line-by-line approach to dialogue management. Instead of having the developer select boolean operators and attribute names, however, the developer would simply type the logic, specify the necessary parameters, and select the resultant function. This highlighted the structure of dialogue control by making it obvious to the developer that each command results in a single function.

With this new design, any necessary instruction could be added to the dialogue manager, and the developer was no longer left helpless with an imposing blank text buffer. It also made things a bit clearer for a developer who wished to change instructions in the dialogue manager, because he or she could see what part of the instruction was being modified, and would know what type of change to make. It was now much simpler to change a single parameter, or the resultant functions. This interface design also had the ordered list of instructions at the far left, where the user could gain a somewhat more intuitive feel for their sequence. While this interface still did not teach the user how to use dialogue management, some of the presentation helped make it a little more intuitive, without explicit instructions.

Add the following line to the file:

This part is designed to help you add lines to the DCTL file editor above.

Part 1:

Select the Predicate:

Logic:	Key Name:	Test:	Number:
	:num_found	>	0
AND	:nth	TRUE	
AND	:selected	FALSE	
Select...		Select...	

Part 2:

Select the resulting function:

filter_db_tlist
-----------------

Part 3:

Specify parameters for function:

filter_key	division
match_condition	strstr
Select...	

Modify DCTL File

Figure 3.3: Dialogue Control v.1

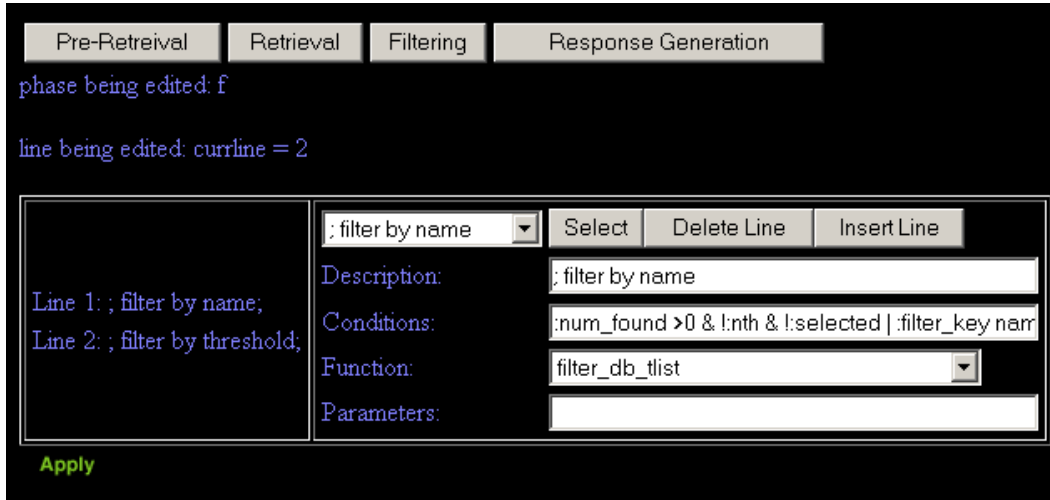


Figure 3.4: Dialogue Control v.2

Another style of editing we explored was completely different. Instead of presenting the user with a structure for adding or modifying instructions in a sequential list, the sequence was hidden completely from the developer. Instead, the user was given a list of resultant functions which would all be incorporated into a 'default' dialogue. Upon selecting one of these functions, a specific interface would appear to guide the developer to correctly specify the corresponding instruction. Although the developer would never actually see the created instruction, he or she would understand that a certain functionality had been added.

Presenting dialogue in this way to the developer signified a large shift in philosophy. The developer no longer had a holistic view of the dialogue control. One would know that certain functionality had been added or taken away from the dialogue, but not how it fit into the flow control of the instruction sequence.

In a sense, this was a very positive result, because instead of focusing on the unnecessary details, a developer could more quickly grasp the concept of dialogue management. At the same time, however, these changes made the dialogue control extremely inflexible. An expert developer would have almost no use for this new dialogue management interface.

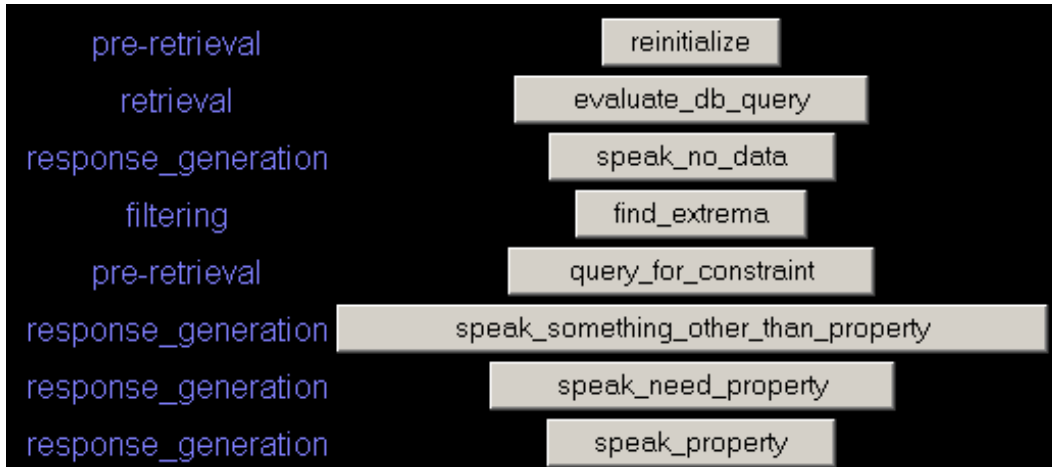


Figure 3.5: Dialogue Control v.3

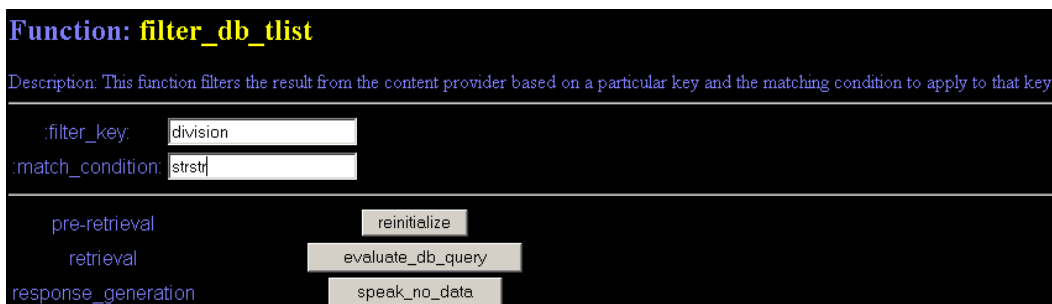


Figure 3.6: Dialogue Control v.3



### 3.2.2 Incorporated DCTL file

As previously stated, dialogue control only involves one configuration file. Therefore, the instinct of an expert developer would likely be to simply edit that file if any changes needed to be made. Forcing an expert developer to use SpeechBuilder to modify one file in a less flexible and less efficient manner would be silly, but leaving a novice developer helpless is also inappropriate. Any developer who has grown accustomed to using a text editor for this purpose would likely resist the requirement to use SpeechBuilder, but a novice has almost no choice. It is assumed that a developer always either modifies configuration files manually or through SpeechBuilder, and we give preference to the manual changes.

One key feature of SpeechBuilder is that it enables the user to create an easily portable domain; everything is contained in a single domain file. Only when this file is compiled are the various configuration files necessary in Galaxy created. In general, however, this domain file is only modified when changes are made through the SpeechBuilder interface. Any modifications made to one of the domain configuration files outside SpeechBuilder are not reflected in the SpeechBuilder-produced domain file, and some of the benefit of using SpeechBuilder is lost as soon as this occurs. The result is that the system, as the developer has configured it, is no longer portable in a single file. In fact, every time a domain is compiled, any modified files are overwritten by the SpeechBuilder-generated configuration files.

We decided that it would be beneficial for a modified dialogue file to be incorporated into the XML domain file upon a SpeechBuilder compile. That means that instead of having the dialogue file be overwritten at compile time, the compiler looks at the file to see if it has been modified more recently than the most recent compile. If it hasn't, then the compile proceeds as expected. If it has been changed more recently, then any changes made inside SpeechBuilder to the dialogue are discarded, and the currently existing dialogue is incorporated into the domain.

This has had a great number of benefits, and very few drawbacks. It gives expert

```

<dialogue>
<evaluate_db_query :query_language espn :destination mars>
<filter_db_tlist :filter_key team_name :db_key teams :match_condition strstr>
<filter_db_tlist :filter_key division :match_condition strstr>
<rank_by_key :rank_key games_back>
</dialogue>

```

Figure 3.7: Dialogue section of Voice Configuration file

developers all the flexibility they want, and here at SLS it has proven very useful. Furthermore, developers do not have to remember to change the permissions of this file after manually changing it, lest it be destroyed during a SpeechBuilder compile. The lone drawback to this functionality has been that if a developer makes manual changes, and then makes changes in SpeechBuilder before compiling, those changes are lost. In practice this has rarely, if ever, occurred.

### 3.2.3 Voice Configuration file

After all of the previously mentioned different dialogue interfaces had been developed, we came to the conclusion that developers with more programming experience might prefer a different style of interface than the graphical web-based SpeechBuilder. This is because SpeechBuilder was essentially optimized for people with both little programming experience and little domain development experience.

#### What is a Voice Configuration file?

At present, a Voice Configuration file (VCFG file) is essentially a list of all the configurable pieces of a domain that could be configured through SpeechBuilder. Instead of naming *attributes* and *actions* in SpeechBuilder, the developer lists them in a VCFG file. Likewise, the developer can list any additional vocabulary. This format is much more concise, because the developer does not have to use several interfaces, and it also allows the developer the ability to see the whole domain at a single glance.

By using VCFG, the developer is now able to see all the configurable aspects of the dialogue at the same time, as shown in Figure 4.2. An example of a full domain

specified in VCFG is the Mars domain, given in Appendix A. Some parts of the dialogue will be the same independent of the domain, and those parts are left to the compiler. Examples of this include server initialization, and the response to the user when there is no data. The configurable pieces of dialogue can be specified in order in VCFG.

In VCFG, it is also much more obvious to a novice developer what is being specified. By placing some commented template lines inside a domain file, the developer can easily determine what changes to make or what lines to add in order to do what he or she wants.

### **3.3 Comparison of VCFG and XML as formats for domain specification**

Originally, XML was chosen as the single-file specification format for a Galaxy domain. At the time, it served all the necessary purposes by storing the information for a domain in a simple and efficient way [6].

One of the primary benefits of XML is its clarity. In general, a developer can look at an XML-formatted file and understand what he or she sees. This is useful for SpeechBuilder because if a developer decides to manually edit the XML representation instead of using the graphical interface, things mesh very smoothly. The representations from SpeechBuilder were designed to be intentionally simple so that this would be easy for developers. It is only when a developer edits a generated file that inconsistencies will begin to arise.

This type of structure is very useful for attributes such as names, example sentences, and vocabulary. There are, however, other parts of a domain which need more than to simply be specified. They must be specified in a specific order.

Using XML to specify attributes in order can be somewhat clumsy. The intent of XML is to contain information in a hierarchical format, and the prevailing algorithms for XML information storage and retrieval do not make use of sequential order. This

makes the algorithms faster and more efficient, but they are not as useful for creating an easy-to-read domain representation, where ordered items are necessary.

Take, for example, a dialogue, which consists of a sequence of 50 instructions. When SpeechBuilder stores this dialogue, it may maintain a representation of the order, but the instructions will not necessarily be written in logical order to the domain file. If a developer then tries to manually edit the XML representation, it is a difficult and painful task – completely contrary to the desired simplicity of a domain file. Instead, SpeechBuilder simply writes all the instructions as a single string in a single element in the XML representation. This, however, does not take advantage of the XML structure, nor does it make it much easier for a developer to edit manually.

The VCFG format has solved these problems, although it has introduced some of its own. By using a new format for domain specification, we are able to make use of all the simplicity of XML, because the visual format of VCFG can be exactly like XML for components where it is appropriate. In fact, by parsing those sections with algorithms that have been developed for XML, we can truly keep the efficiency of that format. For components where XML isn't the most efficient structure, we use our own format, and write our own compiler. Then, instead of compiling the XML representation, we compile the VCFG.

VCFG, however, was designed more as a text-based representation for SpeechBuilder, not as a replacement for XML. It is true that it serves as a better domain file format than XML, because of the reasons mentioned above, but it is not nearly as easy to automatically generate a VCFG file. It is possible to incorporate some of the simplicity of XML formatting into VCFG, but some extra complexity is added, nevertheless. For now, we have used a simple script which converts a file from VCFG to XML, then processes the XML, but we will eventually move to a purely VCFG-based compiler.

Other than these SpeechBuilder-based drawbacks, most aspects of the VCFG have been very positive, because developers with significant domain building experience

have preferred editing a single text file instead of using a graphical interface.

### 3.4 Frame Relay

The most recent addition to the Galaxy architecture is the frame relay. As was explained earlier, it is a server whose purpose is to mediate between the hub and any developer-created external servers. Because it supports interfaces (such as that between the hub and the database server) which the hub already recognizes, it enables the user to replace a given server with his or her own specific server. In order to enable the replacement of other servers, it was necessary to build the frame relay in such a way that it could recognize the function calls that would have been made to the server that it replaces.

For example, the hub might have made a `database.send` call to the database server, and sent an associated frame with query information for the database. With the frame relay in place, it recognizes `database.send` and the format of the frame that arrives with it, and it knows how to pass that data to the user-defined data server. When the data server returns the result frame to the frame relay, it is packaged up in the same way a traditional database server would have done, and the hub receives the information as if the database server itself had performed the exchange.

For my work, the frame relay was used to replace the database server. Because the frame relay can recognize any of the function calls made to the database, it serves as a transparent substitute. That is, the rest of the system does not realize that the original database server has been replaced.

When the frame relay receives a database function as input, it does nothing more than pass the input information on to a third-party server, which is defined by the developer. The developer is responsible for providing a server which returns data to the frame relay in the correct format, because the frame relay will simply take that information and pass it right back to the server that originally called it, as shown in Figure 3.8.

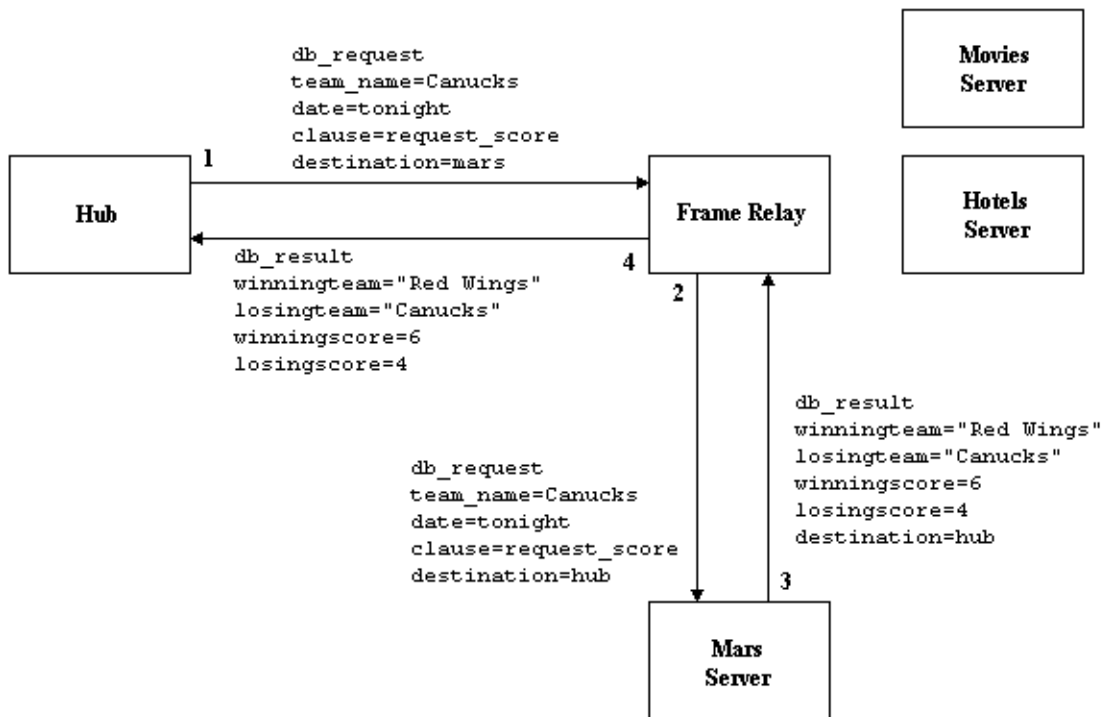


Figure 3.8: Sample transaction between Hub and Frame Relay, with access to external data server

It is useful to note that the frame relay was designed to handle communication with any number of servers. A result of this is that it is not necessary to start a new frame relay and data servers every time a domain is started. In principle, one machine could host the frame relay, and the data servers could be hosted on several machines. As long as each data server contacts the frame relay to notify it of its existence, they will be able to communicate.

Fitting the frame relay into the Galaxy architecture was only the first step. Each data server designed to work in conjunction with the frame relay has a unique name, and the frame relay must be informed of which data server to access in a given situation. This information is specified in the dialogue manager, because the dialogue server is the component of Galaxy that decides when to access a database.

For example, a hotel information domain could specify “:destination hotels” in the dialogue. A beneficial side effect of this structure is that it makes it very easy to use multiple data servers for a single domain. The dialogue manager simply knows when to use which server.

As a matter of practicality, developers should not have to be concerned with the details of replacing the database with the frame relay; this should be handled by the compiler. A special command was created that starts a SpeechBuilder domain using the frame relay instead of the database. By invoking this command, a developer can run the external data servers which the frame relay will need to access.

# Chapter 4

## Results

As a result of this project, dialogue and discourse have been added to SpeechBuilder and to Voice Configuration. This has been done by adding two new interfaces in SpeechBuilder, one for dialogue and one for discourse. Voice Configuration, which is a new file format, also has two separate sections of the file which are for these components.

The addition of these new components makes possible the development of domains which have a higher level of interaction with the user than was previously possible, and which have a higher level of “intelligence.” That is to say that the dialogue helps the system know under which circumstances to ask questions, and the discourse knows which questions have already been answered.

### 4.1 Configuration of Dialogue

#### 4.1.1 SpeechBuilder

Making dialogue control configurable through SpeechBuilder has been the most challenging task of the entire project. This is not because the technology necessary to create the configuration files is complex or difficult to use, but rather because of the non-intuitive nature of dialogue control. The difficulty in presenting dialogue configuration in an intuitive and understandable format for developers has been great. In the end, it was decided to stay with version 2 of the dialogue control interface,



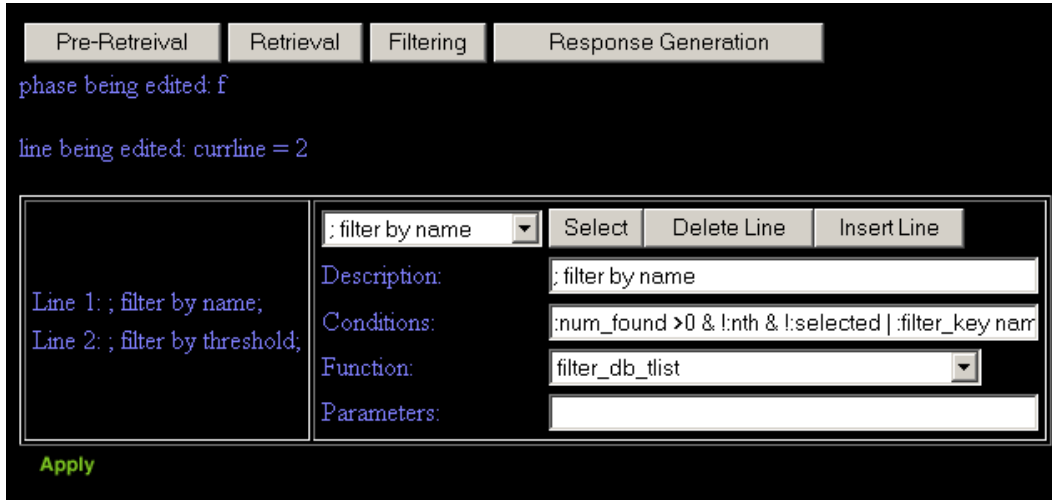


Figure 4.1: Dialogue Control v.2

because it was the last one that had complete functionality. This interface is shown in Figure 3.4.

As shown, this interface has the four phases of dialogue control divided into four separate sections. Although it is not strictly necessary for the developer to follow this framework, it provides a more structured way of approaching dialogue control, and it also enables the developer to focus on a certain purpose of the dialogue at once. To make configurations, the developer first selects one phase of the dialogue. After this selection is made, the list of instructions corresponding to that section will appear in the left section of the interface. This provides a concise, single-glance overview of the section.

### Explanation of the Interface

Once the developer has selected a phase of the dialogue, he or she can then select one of the instructions in the top drop-down box located in the right section of the interface, and click “select.” The selected line will appear in the boxes below in the following format:

**Description:** This is an optional associated comment which gives the developer information about the dialogue instruction. This comment is used for the overall

display on the left side, and is also used for the drop-down box on the right side. By making use of this comment, the developer is able to avoid looking at code when unnecessary.

**Example:** If the purpose of an instruction is to filter the responses from the database by `team_name`, then the description might be “Filter by Team Name.” Again, this is purely for the convenience of the developer, and plays no role in the execution of the instruction.

**Conditions:** This is a boolean expression which indicates when an instruction should be invoked. In this box, the developer can specify attributes, any attributes which are returned from the database, boolean operators, comparative operators, etc. The user can also specify an attribute by itself, indicating that the attribute need only exist (that is, it has some value, and the value doesn’t matter) in the system.

**Example:** “`:property & :num_found =0 --> speak_no_data`” The conditions for this instruction are the existence of `:property`, and `:num_found = 0`, which means that no results are left after retrieval from the database or after filtering. In the example of dialogue state given before, in Figure 3.1, the function `speak_no_data` would not be invoked, because although `:property` does exist (the value is “extension”), `:num_found` equals 1.

**Function:** A drop-down box here lists the different functions that can be executed if the conditions are met. These functions are all pre-compiled in a file that comes with the generic dialogue server.

**Example:** In the Filtering section, the developer might want to select the function `filter_db_tlist` or `rank_by_key`. In the Retrieval section, `execute_db_query` would probably be selected.

Note that these functions are what logically separate one phase of the dialogue control from another. That is to say that there are certain functions which are designed for filtering data from the database, other functions which are designed for retrieving data, etc.

```
<dialogue>
<evaluate_db_query :query_language espn :destination mars>
<filter_db_tlist :filter_key team_name :db_key teams :match_condition strstr>
<filter_db_tlist :filter_key division :match_condition strstr>
<rank_by_key :rank_key games_back>
</dialogue>
```

Figure 4.2: Dialogue section of Voice Configuration file

**Parameters:** Some functions have parameters which must be passed in to work correctly. In fact, these parameters are the bare essentials which must always be specified by the developer. The other three components of an instruction (Description, Conditions, and Function) can all default to some value, but the parameters are necessarily domain specific. In Voice Configuration files, this is the only specification the developer is expected to do.

Example: Function `rank_by_key` requires the parameter `rank_key`, which indicates the key by which to rank the results.

Once the developer understands the nature of dialogue control, it is a simple task to specify the dialogue in SpeechBuilder. The developer selects an instruction and modifies it in the provided boxes. After making any necessary changes or additions, the developer must click “Apply” for the changes to be saved.

### **Additional Commentary**

Truly, the issue of dialogue control configuration was best solved with the emergence of the Voice Configuration file. An appropriate way to proceed for future work might be to create a new dialogue control interface in SpeechBuilder which resembles the interface in the Voice Configuration file. That is, only the configurable instructions are available, but some indications can be given to the user about what the instructions will do, and which parameters are required. This would be a modification of the last, experimental dialogue control interface.

### 4.1.2 Voice Configuration

Configuring dialogue in a Voice Configuration file is conceptually different than configuring dialogue in SpeechBuilder. Instead of the developer having to arrange all of the instructions manually, almost everything is done automatically. The developer never sees anything more than those instructions which have to be configured specifically for a domain, and the developer is free to place those instructions in the most appropriate order.

For example, in Figure 4.2, there are only four configurable instructions listed. Other instructions could also be added in this same format, but it is assumed that the developer has a knowledge of the necessary parameters and available keys that may be in use by the system or the database.

## 4.2 Configuration of Discourse

Adding discourse to the Galaxy configuration utilities is considerably different from adding dialogue, because it does not require nearly as much training for the developer. Whereas helping a developer configure dialogue can be very tricky because of the nature of dialogue, helping a developer use discourse is simple because the concepts are fundamentally simpler. For example, inheritance does not need to be specified in any sequence, but dialogue control does. Attributes that can be inherited include those which are extracted from user utterances, and those which are returned from a data server.

There are two ways in which the discourse of attributes must be configured. First, the developer must specify which attributes are “inheritable.” If an attribute is inheritable, that means that the value can be passed from one contextual frame to another. Examples of inheritable attributes in a Hotels domain might be the name of the city or the brand of the hotel. An attribute that possibly shouldn’t be inherited could be the street address. It makes sense that if a user says to the system, “How



Figure 4.3: Current interface for discourse configuration in SpeechBuilder

about Boston?” after having received the street address for a hotel in Chicago, the system should not limit possible responses to hotels with the same street address (which would likely be none).

The second type of specification to be made by the developer is which attributes are maskable. The concept of masking has already been described in Section /refsec-masking, but the mechanism for specification is as follows: Attributes are specified in sets. With each “masked” attribute, there is a set of “masking” attributes. That is to say, if any one of the masking attributes is updated in the context, then the masked attribute is dropped from the context.

The discourse server makes use of these two types to decide which attributes get passed from one context to the next, and which do not.

### 4.2.1 SpeechBuilder

Discourse configuration in SpeechBuilder is as simple as editing the file manually. The interface is shown in Figure 4.3.

The developer must first select the type of context to be specified (that is, Inheritable Attributes or Maskable Attributes) by choosing the type from the upper drop-down box and selecting the type. Having done so, the list of attributes in that category will appear in the left section of the interface. The developer can either modify existing context specifications or create a new one. In order to modify the masking of a attribute which has already been listed, the developer selects the desired

```

<discourse>
league | team_name -> !city
league | city -> !team_name
team_name | city -> !league
team_name | city -> !selected
team_name | city -> !nth
clause -> !selected
clause -> !nth
</discourse>

```

Figure 4.4: Discourse specification in Voice Configuration file

attribute from the drop-down box in the right section of the interface, and selects it. This causes the name of the selected attribute to appear in the **Mask** text box, and the list of attributes which mask that attribute in the **Input** box.

The two text boxes, **Input** and **Mask** represent the two different parts of each inheritance set. The attribute to be masked is named in the **Mask** box, and the other “input” attributes which cause that attribute to be masked are listed in **Input**. Once the developer has made changes to this set of attributes, he or she clicks “Apply,” at the bottom of the interface, and this saves all changes to the SpeechBuilder XML file. When the domain is next compiled, the saved changes will appear in the generated Galaxy domain.

### 4.2.2 Voice Configuration

In a Voice Configuration file, discourse specification is essentially as simple as it is in SpeechBuilder. The pattern used is more like a logical rule than anything else. An example of masking would be the following:

```
league | team_name -> !city
```

Other such rules are shown in Figure 4.4. In this example, if either **league** or **team\_name** is updated, then **city** is deleted.

Although the masked attribute is at the end of each statement, the flow of information in this type of specification is preferable, because it has an “if-then” feel to

```
*MASKABLE_KEYS

.city
league team_name

.team_name
league city

.league
team_name city

.selected
team_name city
```

Figure 4.5: Example of file which specifies inheritance

it. “If `league` or `team_name`, then not `city`.” Contrast this to the actual inheritance file, where the masked attribute is listed first, and the masking attributes are listed second. An example of this is shown in Figure 4.5.

This is clearly much simpler than the `SpeechBuilder` interface, because it does not require selecting a type, selecting a line, etc. This is ideal, because it is optimized for use by more experienced developers, whereas the `SpeechBuilder` interface is designed to provide a greater measure of explanation.

# Chapter 5

## Applications

To test the functionality of the new interfaces which had been added to SpeechBuilder, along with the Voice Configuration file format, we developed several domains which make use of the new additions. By building these domains, we were able to refine the interfaces we had developed, making them more useful, and we were also able to assure that the functionality was complete.

Three domains are discussed here, namely Echo, Hotels, and Mars. Echo is a simple domain which was developed for the purpose of helping developers learn to use the frame relay. Hotels is a domain which existed before the discourse and dialogue were added to SpeechBuilder, and was meant as a benchmark to show that SpeechBuilder configured these two components correctly. Finally, Mars was built from scratch, to prove that new domains making use of discourse, dialogue, and the frame relay could be built fully within the framework of SpeechBuilder.

### 5.1 Echo

The Echo domain is the simplest of the three domains created to take advantage of the frame relay and the new dialogue. It serves as a “Hello World” domain for developers who are learning to use the frame relay. The function of this domain is to receive an input utterance from the user, and to speak back, “I think you said (*user\_utterance here*).” The Echo server interacts with the frame relay, and behaves



```
...
!:nth --> set_up_selected
!:selected & num_found <0 | :destination echo --> evaluate_db_query
:property & :num_found =0 --> speak_no_data
...
```

Figure 5.1: Excerpt of dialogue control file which specifies “echo” as the data server as a database, where the input utterance is the parameter, and the same utterance is the output. Of course, this does not really access any external data source, it simply takes the input and gives it back.

By exploring the server code, developers can discover how to interact with the frame relay interface, and how to get the results they want. In order for this domain to function correctly, it is also necessary to specify the name of the server (“echo”) in the dialogue manager. This specification occurs in the instruction which results in the execution of `execute_db_query`, as shown in Figure 5.1. This serves to help developers become more comfortable with dialogue and dialogue specification.

## 5.2 Hotels

Before the new dialogue and frame relay functionality were added to SpeechBuilder, several domains already existed that could be built with it. While most of these have very domain-specific components, the Hotels domain was originally designed to be a more complex domain that could be built with SpeechBuilder. As time passed, it drifted away from its SpeechBuilder origin and became more specific and unbuildable with SpeechBuilder, because it made use of newer components, such as dialogue and discourse.

Hotels is a domain where users can call in and make a hotel reservation in a given city [4]. An example dialogue might proceed as shown in Figure 5.2.

The key differences between Hotels and any SpeechBuilder domain were in dialogue, inheritance and data retrieval. By using the Hotels domain as a benchmark,

**System:** "Welcome to Hotel Information. How may I help you?"  
**User:** "I need a hotel in Boston"  
**System:** "What brand?"  
**User:** "Sheraton"  
**System:** "There are nine Sheratons in the greater Boston area. Please be more specific."  
**User:** "I would like to be as close to Logan Airport as possible."  
**System:** "I have the following entries: Airport Sheraton, Boston Sheraton Deluxe, ..."  
**User:** "I also need a pool."  
**System:** "I have the following entries: Boston Sheraton Deluxe, Lexington Sheraton, ..."  
**User:** "I'll take the Boston Sheraton Deluxe."  
**System:** "Which days will you be staying?"  
**User:** "From tomorrow until next Thursday."  
**System:** "The cost is \$149.95 per night. Do you wish to book this hotel?"  
**User:** "Yes."  
**System:** "I've reserved a room for you at the Boston Sheraton Deluxe from April 19, 2002 until April 25, 2002. Do you need anything else?"  
**User:** "No."  
**System:** "Thank you for using Hotels. Goodbye."

Figure 5.2: Sample interaction from Hotel information domain

we could see that by adding these components to SpeechBuilder we would enable it to make domains as sophisticated as Hotels.

First of all, the dialogue control in Hotels is significantly different than what SpeechBuilder could previously create. Instead of using a normal database or a CGI script, it had a very specific script designed to get data from travelocity.com. At the time this script was created, it did not make use of the frame relay; the name of the script had to be specified for the database server.

The dialogue also had to do things like filter results from the database. For example, if a user asked for a hotel in Boston, the database would return a list of all hotels in Boston. It wouldn't be very useful to speak the full list to the user, so instead the system asks for a brand. When the user responds "Sheraton," the system can then filter through the list of Boston hotels, and if the list is sufficiently small it will return the list of Sheratons in Boston. This is an example of a situation where the more sophisticated dialogue helped the user refine his or her search, without forcing a complete re-specification of the query.

Other dialogue functions in Hotels include the ability to know if the user has specified enough information to actually perform a query. If the user has not even given a city name, then the system will not go to the database and ask for a list of all hotels in the world.

As stated earlier, Hotels was using a domain-specific script which had to be called by the database server. This script was rewritten as a data server which could be called by the frame relay. As a result, Hotels is able to have the same functionality as it did with the previous script, but it now works with the SpeechBuilder architecture. As far as the rest of the domain is concerned, this change in database structure is completely transparent, because the interface from the hub to the frame relay is the same as it was from the hub to the database, except for the fact that it must specify a server destination (that is to say, it must say that it wants to get data from the "Hotels" script).

**System:** "Welcome to Mars sports scores. How may I help you?"  
**User:** "Did Boston win their game last night?"  
**System:** "Do you mean the Boston Bruins, the Boston Red Sox, or the Boston Celtics?"  
**User:** "Boston Red Sox."  
**System:** "The Red Sox beat the Blue Jays 10-3."  
**User:** "How about the Yankees?"  
**System:** "The Orioles beat the Yankees 5-4"  
**User:** "What are the standings in the American League East?"  
**System:** "The standings are as follows: Boston Red Sox at 0 games back, New York Yankees at 1.5 games back, Toronto Blue Jays at 2 games back, Baltimore Orioles at 4 games back, and Tampa Bay Devil Rays at 6 games back."  
**User:** "Thanks. Goodbye."  
**System:** "Thank you for using Mars. Goodbye."

Figure 5.3: Sample interaction from Sports information domain

## 5.3 Mars

The Mars domain is a system in which the user can call and ask for the scores of a certain game played in professional sports. However, this could include college or even high school sports as well, if the database were extended to provide that information. The system listens for the keywords such as team names and cities, and phrases which are specified in the Voice Configuration file.

Mars was built as a proof of concept of the new SpeechBuilder and of the VCFG format. To do this, it makes use of all the new components, which include those found in Hotels. This shows that SpeechBuilder is flexible enough that it can build more domains as complex as Hotels; it can build anything using the same components.

A typical exchange might sound as follows:

### 5.3.1 Database Script

As shown in Figure 5.3, the system has access to all of the latest professional sports scores. These scores are all retrieved via a special server which is specific to Mars. This server works in conjunction with the frame relay.

Initially, the Mars server connects to the frame relay and names itself “mars.” Then, it simply idles and waits for a frame to arrive from the relay. Once it receives a frame, it parses out all the relevant information. For the Mars server, this is all found in the frame value called `db_request`.

The relevant information includes the team name, the city where the team plays, the league in which the team plays, and the type of data sought, which could be the score, the standings, or conceivably any other sport-related information. Having determined these values, the server compares the information to a list of teams it holds internally. The three possible scenarios are that the user-specified information corresponds to a single team, to multiple teams, or to no teams at all. Each of these outcomes results in a different response from the server.

If no teams match the input, then the system will return and say that it does not recognize any teams which match the input. It does not leave the user helpless, however; it scans the list again to see if there are any partial matches, and returns any that exist. The user will then know whether the system misunderstood what was originally spoken, or if the system simply does not recognize the team the user specified.

If multiple teams match the values, then the system will likewise return a list of teams that match the input. This was shown in the example given above. When the user asked if “Boston won their game last night,” the system could not uniquely determine which of the three teams located in Boston the user had referred to. Given this, the server returned the names of the three teams, and the system spoke them back to the user.

As of the writing of this document, the system makes no effort to check which of the possible teams played a game. This functionality would help the system make a more intelligent response, but it would also greatly increase the amount of network requests to the data provider.

When a single team matches the user’s input, then the response is much more

interesting. First, the server determines the professional league of the specified team. Then, there are two functions the server could perform, depending on what the user has requested. If the user has asked for a score, then the server returns a list of scores from the given league. If the user has requested the standings, then a list of standings in that league is returned.

### 5.3.2 Scores

When the user has asked for a score, the server retrieves all the games played in the league that day because they are all provided on the same web page. There is no reason to throw them away, since it requires very little to keep them, and they may prove useful. In the future, the server should be modified so that if the team is ambiguous, but the league is unique, this query would still be performed.

After the web page has been retrieved, the scores are parsed out and the information is appropriately processed to create an individual frame for each scheduled game. Each of these frames contains information such as which team is the home team, which team is the winning team, how many points were scored by the winning team, etc. Some information is actually duplicated in these frames (such as scores), but this helps the system provide more intelligent responses to the user. It also makes it possible to answer yes-or-no questions. If the user asks, “Did the Red Sox win last night?” the system can simply check if `winningteam = ‘Red Sox’` and modify the spoken response based on that.

All of these individual “game frames” are packaged into a single, larger frame known as the `db_result`. This, along with some other information, is returned from the Mars server as the result of the database query. An example of a `db_result` frame is shown in Figure 5.4.

When the system receives the list of scores, it then has the information pertaining to all games played in the league on a single day, although the user only requested the game (or games) played by a specific team. This is where the more sophisticated

```

(c reply
 :destination "hub"
 :db_result {c games
  :nfound 9
  :values ( {q game
    :oppscore 9
    :homeid "Cleveland"
    :winningcity "Cleveland"
    :ownscore 5
    :winningteam "Indians"
    :decision "win"
    :awaycity "Minnesota"
    :awayid "Minnesota"
    :score 1
    :homecity "Cleveland"
    :awayteam "Twins"
    :losingcity "Minnesota"
    :time "Final"
    :teams "Twins Indians"
    :hometeam "Indians"
    :homescore 9
    :cities "Minnesota Cleveland"
    :losingteam "Twins"
    :awayscore 5
    :winningscore 9
    :losingscore 5}
    {q game ...

```

Figure 5.4: Sample `db_result` response from the Mars server. The value shown is a “game frame,” representing the baseball game between the Cleveland Indians and Minnesota Twins played April 8, 2002

dialogue comes into play; we make use of the filtering functionality.

One of the values returned as a part of each game is `teams`. It is a string which contains the names of both teams. The dialogue manager then performs a substring-match operation on the team specified by the user and the `teams` value of all the returned games. When this operation is complete, the list of games held by the system contains only those games played by the team in question. Once the system has filtered out the relevant games from the list, it speaks them back to the user. At that point, the user is free to make another query, ask about another team, and so on.

### 5.3.3 Standings

If the user has asked for the standings, the server's response is very similar. With scores, it returns all the scores from a single day; with standings, it returns all the standings in the league. That is to say that it visits the standings URL, extracts the applicable data for all teams in the league, and returns the full list of standings.

In this case, however, the frames are somewhat different. Each frame contains the team name, city, and league division, but it also contains data which is specific to its division: the number of games back. The number of games back for a team is completely dependent on the other teams in its division. Essentially, it depends on the difference between the records of the specified team and the team in the division which has the best record. If they are the same team, then the number of games back is equal to zero; otherwise it is likely greater than zero (unless multiple teams are tied for the division lead).

This data is only useful in the context of a team's division. Therefore, a lot of work is left to the dialogue manager after the results have been returned. Once the system has received the standings data from the Mars server, it must first use the filtering mechanism described above to select only the teams which are in the division of the team specified by the user. The same substring match is executed, but in this



case it compares the division of each team returned from the database to the division of the team specified by the user.

The result of this operation is a list of several teams, all of which are from the same division. Each team is a certain number of games back from the leader, and an appropriate response from the system would read the names of these teams in ascending order of games back. At this point, a `rank_by_key` function which is also provided by the new dialogue is executed. By specifying the attribute which corresponds to the games back, and sorting the list<sup>1</sup>, the system has the list of teams in the order desired by the user, and it is spoken back. The attribute `games_back` is returned from the data server.

One function which could be added to the system would be a wild-card calculator. A wild card team in a league is a team which is not leading its division but whose record is good enough to merit a playoff berth. Determining the wild-card standings, however, is slightly more complicated than division standings, because the desired list of teams is much longer, and the data for each team is still based on the respective divisions.

For this to work, the data server would have to return both a team's games back in their division, and the games back in their conference. This could be calculated by the server itself, based on the actual wins and losses for each team. Once this data was returned to the server, it could be filtered by conference and then sorted by conference games back.

### 5.3.4 Inheritance

The Mars domain exposes some interesting issues concerning attribute inheritance. Typically, if a user asks about the Boston Red Sox, and then later asks "What about the Yankees?" the system should understand that the user does not intend to ask

---

<sup>1</sup>In fact, the information is actually returned from the ESPN server in order, because the standings are displayed in order on the website. To show that the `rank_by_key` functionality worked, it was necessary to scramble the results from ESPN.

about the Boston Yankees. This is an example of masking, where the city in context, Boston, is masked (or ignored) when the user specifies a new team name.

In this particular domain, however, this is not always the appropriate response. If the user asks about the New York Islanders, then says, “How about the Rangers?” it would be silly for the system to respond, “Do you mean the New York Rangers or the Texas Rangers?” because the New York Rangers are another hockey team, and the Texas Rangers are a baseball team. Unfortunately, this is the behavior of the system as it exists now, because as in the previous example, it masks the city information when a new team name is specified. It seems obvious that if a user was first asking about a hockey team in New York, then he or she is probably referring to the other hockey team in New York, given that it is an option. To ask the user if a different team from a different sport is being referred to is not an intelligent response.

I decided that for Mars, although this is an annoying and unintelligent form of response, it was better than the alternative, because people are more likely to ask about different teams in the same sport, and most team names are not duplicated in other sports. For instance, if a user were to ask about the Yankees and then the Mets, the system would respond as one would expect.

At first glance, it appears that it might be possible to mask only the team name while preserving the league, but this will also fail. Under these circumstances, if a user asks about the Boston Red Sox, and then about the Boston Celtics, the system will match no teams because it would search for a team named the Celtics in Major League Baseball.

A better algorithm to handle these issues might have the server do a search beginning with the masked data, and if there were no team match it would do another search without any of the masking. This would introduce a new set of problems, but it would still be an interesting exercise, with the possible benefit of a system that can make much more intelligent responses.

### 5.3.5 Evolution of Mars Server

At first, the Mars Server did almost all the work for the system. The system would submit all the query information, and the server would attempt to determine a unique team. Having done so, it would visit the appropriate URL, and return the score or scores for that team.

From the user's perspective, it had the same functionality then as it does now. The difference from the system perspective is that with the old arrangement, no filtering was necessary; the filtering had effectively been done by the server. Returning all the results from a single day, however, gave the opportunity to take advantage of the new dialogue control which had been added to SpeechBuilder.

Originally, the Mars server could only return information about scores, but later the ability to return standings was added. Because of this, it was necessary to specify different behaviors, depending on whether the system had submitted a *request\_score* or a *request\_standings* operation. It was decided that in the case that the system made a request to the Mars server without a specific type of request, scores would be returned if there was sufficient contextual information to perform the query.

This is important because in many circumstances, the speech recognizer is not going to produce an utterance which has a corresponding action, although the information necessary may be available. In fact, in some cases it can take several tries for a user to give an utterance which actually corresponds to an action, and this would become extremely frustrating. Therefore, it is important to at least make a best-effort guess and return some data, even if the system later determines that it is not needed.

# Chapter 6

## Conclusion

The additions made to SpeechBuilder and the different compilers have made it possible to easily create more sophisticated domains. Although it was previously possible to create such domains as Hotels and Mars, it was not simple, and was unrealistic for novice developers.

Hotels is a domain that existed before SpeechBuilder could create domains with such a dialogue manager. The changes made to SpeechBuilder were meant to bring it to the point where it could produce domains like Hotels. Now, this domain can be produced by using SpeechBuilder or Voice Configuration files.

As a way of proving the new system, Mars was created using only a Voice Configuration file and a special data server. This type of domain could not previously have been made using SpeechBuilder. The most remarkable part of the Mars domain is what it can accomplish using only the features available in a Voice Configuration file.

As a result of this project to bring sophisticated dialogue to the novice developer, developers also benefit in three additional ways: Voice Configuration files, dialogue control file incorporation to XML, and the frame relay.

In practice, the Voice Configuration file format has proven beneficial and useful for developers at SLS. Given that a goal has been to bring domain creation to developers who might not otherwise have an adequate background, VCFG also seems to meet these needs because of its simplicity and flexibility.

Granted, a developer can not create as complex a dialogue using VCFG as can be done manually editing the dialogue in a text editor, but by using this new format a new developer is able to quickly grasp the most relevant configurable concepts of dialogue configuration. At the same time, the sophistication of dialogue available to a developer using this format is sufficient to create domains such as Mars or Hotels.

Developers using SpeechBuilder also benefit because of the new ability to have modified dialogue files incorporated into the SpeechBuilder XML representation. In my opinion, this type of functionality should continue to be added to SpeechBuilder, because it promotes the portability of domains. I feel that this has been one of the great, yet simple, ideas to result from this project.

Using the frame relay enables developers to customize the data retrieval in a domain by controlling exactly where data is retrieved. By using this new abstraction, a developer can still make use of a relational database, but has the option of using any source necessary.

Although a developer may not be able to easily create a Galaxy domain using the very newest technologies, more and more sophisticated domains are becoming accessible for novice developers through interfaces such as SpeechBuilder and Voice Configuration. Eventually, such interfaces will likely be the prevailing method for domain developers, so it is crucial that they continue to incorporate the best available functionality.

# References

- [1] Zue, V., Glass, J.; “Conversational Interfaces: Advances and Challenges,” *Proc. IEEE*, 2000.
- [2] Seneff, S., Lau, R., Polifroni, J.; “Organization, communication, and control in the Galaxy-II conversational system,” *Proc. Eurospeech*, Budapest, Hungary, 1999.
- [3] Glass, J., Weinstein, E.; “SpeechBuilder: Facilitating Spoken Dialogue System Development,” *Proc. Eurospeech*, Aalborg, Denmark, 2001.
- [4] Polifroni, J., Chung, G.; “Promoting Portability in Dialogue Management,” *Submitted to ICSLP 2002*,
- [5] Seneff, S., Polifroni, J.; “Dialogue Management in the Mercury Flight Reservation System,” *Proc. ANLP-NAACL 2000, Satellite Workshop*, pp. 1–6, Seattle, Washington, May, 2000.
- [6] Weinstein, E., “SpeechBuilder: Facilitating Spoken Dialogue System Development,” M.Eng. Thesis, Massachusetts Institute of Technology, May 2001.
- [7] Filisko, E., “A Context Resolution Server for the GALAXY Conversational Systems,” M.S. Thesis, Massachusetts Institute of Technology, May 2002.

# Glossary

**Conversational System:** A speech interface through which a user can access information from a database. This type of speech interface ideally does not instruct the user concerning what to say, but rather allows the user to direct the discussion by conversing naturally with the system.

**DCTL:** The name of the file format used to control a Dialogue Server. Short for Dialogue ConTroL.

**Dialogue:** In general, a conversation between two or more persons. With respect to a conversational system, it is the conversational element between the user and the system.

**Dialogue Control or Management:** The component of a conversational system which determines how the system will respond to the user. This includes preparing for a database query, actually sending queries to the database, filtering the results from the database, and determining how to present the results to the user.

**Dialogue Instruction:** A single line in a DCTL file; this typically includes (1.) a boolean predicate which is dependent on the existence or state of certain values, (2.) a list of parameters which are necessary for the resultant function, and (3.) a dialogue function which executes if the boolean predicate is satisfied.

**Discourse:** The context of the conversation between the user and the system. This is the component of the system which remembers what the user has said, what has been retrieved from the database, and what values have been assigned by the system.

See also **Inheritance** and **Masking**.

**Domain:** A conversational system which is an expert in a certain realm of information. Examples could be a sports domain, a flight information domain, or a weather domain.

**Frame Relay:** A new component which can replace previous Galaxy servers by passing frames to and from another, developer-created, server. The example in this thesis has the Frame Relay replacing the database server, and the developer has created an external data provider which communicates with the Frame Relay. Therefore, when any of the other Galaxy components think they are communicating with the database, they are actually communicating with the Frame Relay, but the interface is the same.

**Galaxy:** The architecture for conversational systems developed at SLS.

**Generic Dialogue Server:** This replaces the need for a developer-created, domain-specific Dialogue Server. Several generic functions are made available for the developer; these provide for most of what the developer needs.

**Hotels:** A conversational system domain which pertains to hotel reservation information.

**Inheritance:** The idea that certain information should be retained between statements by the user. For example, the user asks for a hotel in Boston, then later says he or she needs a hotel with a pool. The system already knows the user is looking for a hotel in Boston because that information was inherited from a previous statement. As shown, this is what makes it unnecessary for the user to re-specify all information in any given statement. This is the default behavior. See also **Discourse**.

**Mars:** A conversational system domain which pertains to sports scores and standings.

**Masking:** The opposite of **inheritance**. This is necessary when two attributes may be linked in a certain way. For example, a user asks about the Boston Red Sox, then



asks about the Yankees. The system knows not to inherit the city Boston, because that information has been **masked** by the system. This is not the default behavior, and each instance where masking is necessary must be explicitly specified by the developer. See also **Discourse**.

**SLS**: Acronym for Spoken Language Systems, a research group in the Laboratory for Computer Science at MIT.

**SpeechBuilder**: A utility which aids developers in creating conversational systems by automating certain tasks.

**VCFG**: A format for specifying a conversational system domain. A domain which is specified in **VCFG** format must be compiled, which creates a full Galaxy domain.

**XML**: Acronym for e**X**tended **M**arkup **L**anguage. Used in this context as a format for specifying a conversational system. As with **VCFG**, an **XML**-specified domain must be compiled into a full galaxy domain.

# Appendix A

## Mars Domain

What follows is the Voice Configuration file source from which the Mars domain is compiled.

```
<property>
<language> = English
</property>

<constraints>
<request_score> = (league | team_name | city)
    {prompt_for_team_name_city};
</constraints>

<dialogue>
<evaluate_db_query :query_language espn :destination mars>
<filter_db_tlist :filter_key team_name :db_key teams
    :match_condition strstr>
<filter_db_tlist :filter_key division :match_condition strstr>
<rank_by_key :rank_key games_back>
</dialogue>

<discourse>
league | team_name -> !city
league | city -> !team_name
team_name | city -> !league
team_name | city -> !selected
team_name | city -> !nth
clause -> !selected
clause -> !nth
```

</discourse>

<actions>

<request\_opponent>

= who was the Mariners opponent  
| who (did | do) the Red Sox play  
| who's playing the Red Sox

;

<goodbye> = good bye | bye\_bye | bye | bubba now  
| see you | see you later;

<request\_score>

= Who won the Mariners game  
| Who's winning the Mariners game  
| (What | How) about the Mariners  
| (What | How) about Seattle  
| What (is | was) the Mariners score  
| Give me the Mariners score  
| I need [to | know] the Mariners score  
| [(would | could) you] [please] tell me the Mariners score  
| What (is | was) the score for the Mariners game  
| Who won the Boston game  
| Did Boston win their game  
| [how about] [the] Mariners game

;

<request\_standings>

= What are the standings in the A L East  
| Give me the standings  
| How about the standings

;

</actions>

<response>

welcome "Welcome to the Mars Sports Scores system.

How may I help you?"

score "The :winningteam beat the :losingteam  
:winningscore :losingscore"

winner "The :ownteam beat the :oppteam :ownscore :oppscore"

loser "The :ownteam lost to the :oppteam :oppscore :ownscore"

div\_standings "The standings are as follows :standings"

opponent "Their opponent is the :oppteam"

gametime "The game will be played at :time"

goodbye "Thank you for using Mars. And remember to watch

SportsCenter every night at 11 10 central only  
on E S P N!"

</response>

<attributes>

<team\_name> = Angels | Astros | Athletics | Blue Jays | Braves |  
Brewers | Cardinals | Cubs | Devil Rays | Diamondbacks | Dodgers |  
Expos | Giants | Indians | Mariners | Marlins | Mets | Orioles |  
Padres | Phillies | Pirates | Rangers | Red Sox | Reds | Rockies |  
Royals | Tigers | Twins | White Sox | Yankees | Bruins | Avalanche |  
Canucks | Maple Leafs | Oilers | Senators | Blackhawks | 49ers | 76ers  
| Bears | Bengals | Bills | Blue Jackets | Blues | Broncos | Browns |  
Buccaneers | Bucks | Bullets | Bulls | Burn | Canadiens | Capitals |  
Cavaliers | Celtics | Chargers | Chiefs | Clippers | Colts | Cowboys |  
Coyotes | Crew | Devils | Dolphins | Eagles | Earthquake | Falcons |  
Fire | Flames | Flyers | Galaxy | Grizzlies | Hawks | Heat | Hornets |  
Hurricanes | Islanders | Jaguars | Jazz | Jets | Kings | Knicks |  
Lakers | Lightning | Lions | Magic | Mavericks | MetroStars | Mighty  
Ducks | Nets | Nuggets | Pacers | Packers | Panthers | Patriots |  
Penguins | Pistons | Predators | Raiders | Rams | Rapids | Raptors |  
Ravens | Red Wings | Redskins | Revolution | Rockets | Sabres | Saints  
| Seahawks | Sharks | Spurs | Stars | Steelers | Suns | Supersonics |  
Thrashers | Timberwolves | Titans | Trailblazers | United | Vikings |  
Warriors | Wild | Wizards;

<city> = Anaheim | Arizona | Atlanta | Baltimore | Chicago |  
Cincinnati | Colorado | Detroit | Florida | Houston | Kansas City |  
Los Angeles | Milwaukee | Minnesota | Montreal | New York | Oakland |  
Philadelphia | Pittsburgh | San Diego | Tampa Bay | Texas | Boston |  
Cleveland | San Francisco | Seattle | Toronto | St Louis | Buffalo |  
Calgary | Carolina | Charlotte | Columbus | DC | Dallas | Denver |  
Edmonton | Golden State | Green Bay | Indiana | Indianapolis |  
Jacksonville | Memphis | Miami | NYNJ | Nashville | New England | New  
Jersey | New Orleans | Orlando | Ottawa | Phoenix | Portland |  
Sacramento | San Antonio | San Jose | Tennessee | Utah | Vancouver |  
Washington;

<league> = Major League Baseball {mlb} | Major League Soccer {mls} |  
National Basketball Association {nba} | National Football League {nfl}  
| National Hockey League {nhl};

<property> = date | opp\_score | opponent | own\_score | lost {score} |  
won {score} | lose {score} | win {score} | homescore | awayscore |  
awayteam | hometeam | underdogteam | favoredteam | losingteam |  
winningteam | decision | pointspread | overunder | standings

```
{standings};  
<division> = A L East {AL East} | A L West {AL West} | A L Central {AL  
Central} | N L East {NL East} | N L West {NL West} | N L Central {NL  
Central};  
</attributes>
```

# Appendix B

## List of files generated by SpeechBuilder

The following is a list of all files which are generated by a SpeechBuilder compile of the Mars domain. In general, the word “mars” would be replaced by the name of the domain which is compiled.

Each of these files is used to configure one of the nine servers used in systems built with the Galaxy architecture.

```
112.brec
D-service.tcl
bpcs.trim
database.dctl
domain.info
frame.cmd
frame.sas
local.cmd
mars.baseforms
mars.baseforms.fst
mars.blabels
mars.bmodels
mars.bpcs
mars.classes
mars.dctl
mars.ffst
mars.rec
mars.sents
```

mars.sfst  
mars.tsents  
mars.vocab  
mars.vocab.last  
mars.wconf  
mars.wlex  
mars.xml  
menu  
models  
nl.txt  
oxclass.cmd  
oxygen.cmd  
sb.sas  
servers  
speechbuilder-common-nobackend.pgm  
speechbuilder-common.pgm  
speechbuilder-nosynth.pgm  
speechbuilder\_rec.txt  
start.cmd  
stop.cmd  
timestamp  
catalogs/mars.cat  
catalogs/cgi/mars.cgi.mes  
catalogs/cgi/mars.cgi.rul  
catalogs/cgi/mars.cgi.voc  
catalogs/dialogue/mars.eform.mes  
catalogs/dialogue/mars.eform.voc  
catalogs/english/mars.english.mes  
catalogs/english/mars.english.voc  
catalogs/english/mars.rewrite.rul  
catalogs/local/mars.english.mes  
catalogs/sql/mars.sql.mes  
catalogs/sql/mars.sql.rul  
grammars/DISCOURSE\_DOMAIN\_DEFS  
grammars/DOMAIN\_DEFS  
grammars/mars.actions  
grammars/mars.constraints  
grammars/mars.inherit  
grammars/mars.tina  
grammars/mars.translations  
pgm/audio.pgm  
pgm/batch\_hotels.pgm

pgm/batch\_rec.pgm  
pgm/db\_query.pgm  
pgm/db\_query\_two.pgm  
pgm/generic\_as\_hotels\_batch.pgm  
pgm/gui.pgm  
pgm/main\_bottom.pgm  
pgm/main\_top.pgm  
pgm/mercury.pgm  
pgm/rec.pgm  
pgm/response.pgm  
pgm/servers  
pgm/speechbuilder.pgm  
pgm/synthesis.pgm  
pgm/system\_error.pgm