# A Context Resolution Server for the GALAXY Conversational Systems

by

Edward A. Filisko

B.S., Computer Science and Linguistics
University of Michigan, Ann Arbor (1999)

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

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
May 16, 2002

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Stephanie Seneff
Principal Research Scientist
Thesis Supervisor

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

# A Context Resolution Server for the GALAXY Conversational Systems

by

Edward A. Filisko

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2002, in partial fulfillment of the
requirements for the degree of
Master of Science

## Abstract

The context resolution component of a spoken dialogue system is responsible for interpreting a user's utterance in the context of previously spoken user utterances, system initiatives, and world knowledge. This thesis describes a new and independent Context Resolution (CR) server for the GALAXY conversational system framework. The server handles all the functionality of the previous CR component in a more generic and powerful manner. Among this functionality is the inheritance and masking of historical information, as well as reference and ellipsis resolution. The new server, additionally, features a component which attempts to reconstruct the intention of a user in the case of a robust parse, in which some semantic concepts from an utterance appear as unrelated fragments in the knowledge representation. A description is also provided of how the new CR server is being utilized by SPEECHBUILDER, a web-based software tool facilitating the development of spoken dialogue interfaces to applications. We also present an evaluation, in which we compare the performance of the new CR server to the performance of the old CR component.

Thesis Supervisor: Stephanie Seneff
Title: Principal Research Scientist

# Acknowledgments

I would like to extend my gratitude to my thesis advisor, Stephanie Seneff, for her invaluable guidance throughout this research, and for honoring my frequent requests to update the parser in order to make context resolution run more smoothly.

I would also like to acknowledge the members of the Spoken Language Systems Group for their daily support. Specifically, I would like to thank Joe Polifroni for his substantial help in the initial stages of this project, and for his subsequent assistance in my understanding of discourse and dialogue. I also want to thank Scott Cyphers for his help whenever I had problems with the GALAXY code. Additional thanks go out to Grace Chung, Jim Glass, and Eugene Weinstein for their ideas and helpful testing of the server.

I would like to thank my fellow graduate students for bearing with me the several times I attempted to elucidate the difference between discourse and dialogue.

I want to give a special shout out to my office mates for making this an awesome and unforgettable year at MIT! Brooke, thanks for keeping us in line in our crowded, yet homey, office. Xiaolong, thanks for all your advice and for letting me teach you all those great idioms. And Vlad—sausage, pepperoni, and taekwondo…thanks, dawg.

Last, but never least, I want to thank all my family and friends, whose words and deeds were inspirational and always appreciated. Manpreet K. Singh, MD, here's to the stick-to-itiveness of grad students! Andy, thanks for the *ceol*, *damhsa*, *agus craic.* Mom and Dad, thankyou always for your constant encouragement and unconditional support.

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

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Consider the deceivingly simple utterance[1], "Show it to me." This phrase can convey a seemingly infinite number of meanings depending on the speaker, the addressee, the object or action being requested, the accompanying hand or facial gestures made by the speaker, and even on the manner in which the utterance is delivered.

In human-human dialogue, a participant may draw upon the history of what has been said, physical and temporal context, inference, shared world knowledge, and even common sense to interpret and, if necessary, disambiguate another dialogue participant's utterance.

This process, which we call *context resolution*, is ongoing in the course of a human-human dialogue. People carry out this process so automatically, and so effortlessly, that it often appears to be a completely unconscious phenomenon.

Nevertheless, extreme cases of ambiguity can be difficult, if not impossible, to correctly interpret, without further clarification. This is rather eloquently evidenced in the following quotation by Homer Simpson [37]:

"Now listen very carefully. I want you to pull on the *thing*, that's near the other *thing*."

It is not too surprising when Marge proceeds to pull the wrong *thing*, and a burst of flame hits Homer's head—context resolution is, indeed, very important.

[1]The term, *utterance*, is used in this thesis to refer to any speech or written text that may serve as input to a spoken dialogue system.

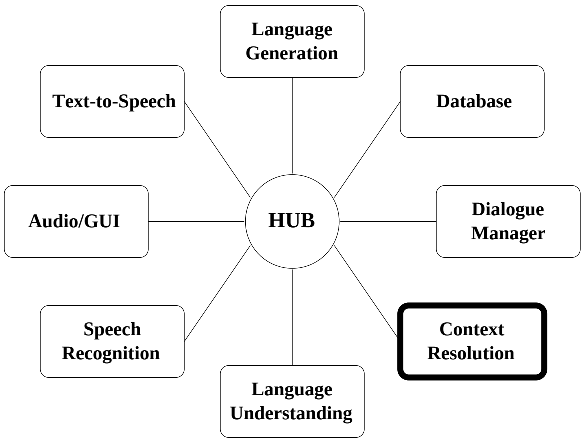


Figure 1-1: A typical GALAXY configuration for a spoken dialogue system. The Context Resolution (CR) server is the focus of this thesis.

In human-computer dialogue, a computer often has no access to physical context and limited access to the other resources mentioned above. This is why the problem of computational context resolution is such a significant challenge.

In the multimodal GALAXY conversational system [2] framework [43, 20], developed by the Spoken Language Systems Group (SLS) at the Massachusetts Institute of Technology (MIT), the problem of context resolution has always been handled within the natural language server. This embedding of functionality, however, has inhibited a generic and extensible representation for context resolution. The framework needed an independent server to handle context resolution in any domain. This is what we have developed and is what will be described in this thesis.

A brief overview of the GALAXY architecture will provide an idea of how the new Context Resolution (CR) server fits into the framework. Figure 1-1 shows a typical GALAXY configuration consisting of a central programmable hub, which handles the communications among various human language technology services via a frame data structure. The audio server captures a user's speech and sends it to the speech recognizer [16], which creates a word graph of hypotheses. This information is dispatched to the language understanding service, via the hub, where the word graph is parsed by TINA [40]. The best hypothesis is encoded in a semantic frame representation and

[2] *Conversational system* and *spoken dialogue system* will be used interchangeably in this thesis.

is dispatched to the CR server. The dialogue manager receives the context-resolved semantic frame and communicates with the database and language generation [3] services to provide an appropriate reply to the user. This response is then audibly realized by the text-to-speech server [52].

In this introductory chapter, we begin with a more detailed discussion of the motivations behind the development of the new CR server, and what designs were employed to fulfill our motivational goals. Then, several example dialogues will be presented to provide an idea of specific phenomena that are handled by the CR server. We close with a brief outline of the remainder of the thesis.

## 1.1 Motivation and Design

In the original GALAXY configuration, context resolution support is provided by the natural language (NL) server. The duties of the NL server, however, are sufficiently numerous and complex without the additional task of context resolution. Although Figure 1-1 shows *Language Understanding*, *Context Resolution*, and *Language Generation* as separate services, they are all embedded in the NL server in the original configuration.

For understanding, the NL server utilizes TINA, which probabilistically parses a user's speech. GENESIS-II [3] handles every aspect of language generation from formatting database queries to generating viable replies to a user. The context resolution code, embedded in the NL server, is as domain-independent as possible, obtaining domain-*dependent* constraints from external files. This context resolution component, henceforth "the old CR component," is extremely useful and has, in fact, worked very well across several domains.

Through our experience with the old CR component, however, we have come to discover several of its shortcomings. First, it displays a significant amount of domain-dependent code and it is not very extensible. This component also does not support context resolution based on key-value pairs, which are the basis for knowledge representation in SPEECHBUILDER [19], a new software tool developed by SLS, which requires key-value based context resolution. Finally, the old CR component lacks a role in the resolution of the robust parse of a user's utterance, in which context resolution would seem to play a part.

Therefore, a new CR component was motivated by the need for a domain-independent and extensible module, a component that supported context resolution based on key-value pairs, and a module that attempted to reconstruct a user's intention in the case of a robust parse. In order to achieve these motivational goals, the existing context resolution functionality would need to be extracted from the NL server and re-realized in a new and independent CR server.

The design of the new CR server was crucial for the successful realization of our motivational goals. For each goal, various implementations were considered until one was found that fulfilled the intended goal and showed potential for facile future modification. Each motivational goal, and related issues, will now be explained.

### 1.1.1 Domain-Independence

Although the initial intention for the old CR component was to be domain-independent, the code displays a fair amount of domain-*dependence*. This is the result of several years of "quick fixes" and code which lent itself to neither simple nor generic extensions. Most of the domain-dependent code handles idiosyncrasies of our most mature and widely used domains; however, one must not lose sight of generality and extensibility in the face of proper function.

To achieve domain-independence in the new CR server, we would have to ensure that the code not contain any information to handle special cases within a given domain. This means that the functions within the CR algorithm would have to show great flexibility in terms of the current domain of the dialogue. All domain-*dependent* constraints, therefore, are specified by the system developer in an external file unique to each domain. This domain-independence will also facilitate the instantiation of the CR component for a new domain, requiring only the completion of some external tables. We also realized that it would be desirable, for the developer, if the constraint specification were as intuitive as possible. Excerpts from this intuitively-aware constraints file will be explained throughout the thesis in great detail.

### 1.1.2 Extensibility

It is evident in many current conversational systems that the division of context resolution labor between the discourse component and the dialogue manager is rather arbitrary. The modification of functionality among the two components can occur rather frequently. Likewise, the sequence of function execution within a context resolution component is often experimental. Such modifications to the division of context resolution labor and function sequence are difficult to perform when the function sequence is controlled by the order of procedure calls in the code. Any change requires an undesirable recompilation of the code; these difficulties arise in the old CR component.

The functionality assigned to the new CR server is based on our previous experience with dialogue systems and human-computer interaction. We have encapsulated each function in the CR algorithm as fully as possible, so that any subsequent function modification may be easily accomplished, ideally, without any significant changes to other functions. Additionally, we have put the CR algorithm function sequence under external control so that recompilation is not necessary to modify the order of execution.

### 1.1.3 Intention Reconstruction

In a spoken dialogue system, a significant additional burden is placed on the understanding component due to spontaneous speech as well as recognition errors. Such cases often result in robust parses, in which only fragments of the user's utterance appear in the semantic frame representation. In the old CR component, no further attempt is made to link these fragments. If such an attempt were made, however, it might result in a possible reconstruction of the user's intention.

The new CR server is responsible for two duties. The first is to interpret a user's utterance in the appropriate context by consulting the history record and, perhaps, world knowledge. The second duty is to attempt to reconstruct a user's intention in the event of a robust parse. We have developed a mechanism in the new CR server, whereby an attempt is made to recover from some recognition and parsing errors. This is accomplished by linking, otherwise unrelated, fragments in the semantic frame. In doing so, the CR server is trying to capture the original intention of the user.

### 1.1.4 Key-Value Support

SPEECHBUILDER is a web-based software tool developed by SLS, which allows the non-expert developer to easily and quickly build a spoken dialogue interface to an application. The key-value-based knowledge representation of SPEECHBUILDER requires a CR component also based on keys and values. The old CR component does not provide such a capability, since its functionality is based on a knowledge representation featuring hierarchical frames.

The new CR server has been designed to support context resolution using both the traditional hierarchical frame representation, as well as the new key-value-based knowledge representation. The complete range of context resolution functionality is not yet provided for the key-value representation. However, significant capabilities do exist, and further research continues to increase CR support in this new representational framework.

Each function within the context resolution algorithm will be described in great detail in Chapter 4. First, however, it is important to be familiar with the type of phenomena that are relevant to the CR server. Several examples of context resolution from our GALAXY domains will be presented in the next section in order that the reader may understand why the CR server is necessary in the GALAXY framework.

## 1.2 Relevant Phenomena

The following dialogues contain several utterances in which context resolution must be performed to obtain a correct interpretation. Each dialogue takes place between

U(1): Show me the beaches in Boston.
V(1): *Here is a map and list of beaches in Boston...*

U(2): Show me the first one.
V(2): *Here is a map displaying Carson Beach...*

U(3): What libraries do you know in Boston?
V(3): *Here is a map and list of libraries in Boston...*

U(4): How do I get to that beach from this library <click on JFK library>?
V(4): *Here are directions to Carson Beach from the JFK Library...*

U(5): Now can you show me universities in the Cambridge area?
V(5): *Here is a map and list of universities in the Cambridge area ...*

U(6): How do I get there <click on Harvard> from here <click on MIT>?
V(6): *Here are directions to Harvard University from MIT...*

Figure 1-2: Sample dialogue between a user (U) and VOYAGER (V), demonstrating the resolution of both spoken and clicked references.

a user and our spoken dialogue system in one of the GALAXY domains, which include JUPITER [53] for weather information, MERCURY [44] for flight reservations, ORION [41] for online task delegation, PEGASUS [54] for flight arrival and departure information, and VOYAGER [17] for Boston area traffic and landmark information.

Consider the dialogue in Figure 1-2 between a user (U) and the VOYAGER city guide system (V). In U(1), the user inquires about beaches. When he asks to see "the first one" in U(2), the system uses the history of the dialogue to interpret this as "the first *beach*."

In U(3), the user switches the dialogue focus to libraries. In U(4), the user makes a reference to "that beach." The only beach that was ever in focus is "Carson Beach" which was mentioned two utterances back. The system must correctly interpret this reference. In U(4), the user also makes a reference to "this library" by clicking on a map. The user makes additional click-references in U(6) while saying, "here" and

U(1): Can you tell me about the churches in Boston?
V(1): *Here is a map and list of churches in Boston…*

U(2): How about museums?
V(2): *Here is a map and list of museums in Boston…*

U(3): How about Cambridge?
V(3): *Here is a map and list of museums in Cambridge…*

Figure 1-3: Sample dialogue between a user (U) and VOYAGER (V). Semantic relationships are used for context resolution.

"there," referring to "MIT" and "Harvard," respectively. The system must correctly interpret such *deictic* references, which are a type of verbal pointing. For example, it would be undesirable for "here" to be resolved as "Harvard" and for "there" to be resolved as "MIT."

In the next dialogue, shown in Figure 1-3, the user (U) converses with VOYAGER (V) using *elliptical phrases*, or *fragments*, from which the omitted information is assumed by the speaker to be understood by the listener. In U(1), the user asks about "churches in Boston." When the user says, "How about museums?," in U(2), he likely means "Can you tell me about museums in Boston?," since "churches" and "museums" have the same semantic class of "buildings." In U(3), the user may mean, "Can you tell me about Cambridge?" or "Can you tell me about museums in Cambridge?" The latter is more likely since "Boston" and "Cambridge" are both cities. VOYAGER must be able to make such decisions based on semantic class.

The next dialogue, shown in Figure 1-4, is between a user (U) and the JUPITER weather information system (J). This dialogue demonstrates how world knowledge may be needed to properly interpret an utterance in context. In U(1), the user asks about the weather in "Cambridge," which is an ambiguous city. JUPITER chooses to alert the user of the ambiguity, but also chooses to provide only a single weather report, which, by default, is for "Cambridge, England." In J(2), JUPITER uses

| | |
|---|---|
| U(1): | What is the weather like in Cambridge? |
| J(1): | *I know of Cambridge, England and Cambridge, Massachusetts. In Cambridge, England, cloudy skies, high near 55. What else?* |
| U(2): | What about Massachusetts? |
| J(2): | *In Cambridge, Massachusetts, partly sunny, high in mid 50s. What other information can I give you?* |
| U(3): | What about Vermont? |
| J(3): | *I know of the following places in Vermont: Burlington, Montpelier, and Rutland. Please select one.* |
| U(4): | Montpelier. |
| J(4): | *In Montpelier, chance of morning snow or rain showers, high in mid 40s. Is there anything else?* |
| U(5): | Goodbye. |
| J(5): | *Thank you for calling!* |

Figure 1-4: Sample dialogue between a user (U) and JUPITER (J). World knowledge is used for context resolution.

world knowledge to interpret U(2) as "What is the weather like in Cambridge, Massachusetts" since there is, in fact, such a place. However, in J(3), JUPITER interprets U(3) as, "What is the weather like in Vermont." If there were a "Cambridge, Vermont," that JUPITER knew about, J(3) would provide the weather for "Cambridge, Vermont," instead of prompting the user to choose a city in Vermont. JUPITER has, in fact, initiated a subdialogue in J(3) to determine the user's intended city. When the user replies in U(4), JUPITER must interpret this fragment as the city for which the user wants the forecast.

The next dialogue, shown in Figure 1-5, is between a user (U) and the PEGASUS flight information system (P). In this dialogue, the user offers several fragment responses, which must be interpreted in the context of the corresponding question posed by PEGASUS. In U(2), "Boston" must be recognized as an arrival city, rather than as a departure city. In U(3), PEGASUS must interpret "Around five pm" as

U(1): What United flights from Denver are arriving today?
P(1): *What city does the flight arrive in?*

U(2): Boston.
P(2): *Can you provide the approximate departure or arrival time?*

U(3): Around five pm.
P(3): *There are two flights. Flight 998 is scheduled to arrive at 4:44 pm, and Flight 1096 is scheduled to arrive at 5:02 pm. Please select one.*

U(4): 998.
P(4): *United flight 998 from Denver is expected in Boston at 4:47 pm local time. It is three minutes behind schedule. Is there anything else?*

U(5): What was the departure time?
P(5): *United flight 998 departed from Denver at 11:21 am local time. What else?*

U(6): That's all.
P(6): *Thank you for calling. Have a nice day!*

Figure 1-5: Sample dialogue between a user (U) and PEGASUS (P), demonstrating the resolution of several elliptical phrases.

either a departure or arrival time. In this case, it simply chooses to accept it as an arrival time. Numbers, in general, are extremely ambiguous. The number in U(4), for example, could be a flight number, an identification number, a portion of an address, or a price, among many other possibilities. In the context of this dialogue, however, it should properly be interpreted as a flight number. In U(5), the user makes use of an elliptical phrase from which he omitted the information, "of flight 998." PEGASUS must realize that "the departure time" refers to that of "flight 998," and not to that of "flight 1096" or any other flight.

The last dialogue, shown in Figure 1-6, is between a user (U) and the MERCURY flight reservation system (M), and demonstrates a rich variety of context resolution phenomena. In U(1), the user expresses his intention to book a trip. In U(2), the user provides an airline and a departure time as fragments. MERCURY must combine this information with that provided by the user in the last utterance. In M(2), MERCURY has found a flight and asks for the user's approval. In U(3), the user does not directly answer MERCURY's query, but instead, enters into a subdialogue to determine if this flight is, in fact, appropriate. Utilizing ellipsis, the user wants to know "the arrival time (of the flight connecting in Dallas Fort Worth leaving at 2:08 pm)." In U(4), the user still desires more information and asks if "it" serves dinner. This pronoun must be resolved by the CR server to be "American flight 3742." MERCURY replies in M(4), but does not pose any further question to the user. The user takes initiative and says, "book this flight," intending for MERCURY to book "flight 3742."

In M(5), MERCURY books the flight and inquires about the date of return. The user provides a relative temporal reference of "four days later" which must eventually be resolved to the absolute date, "June 21." After being given some options based on the connection city, the user utters the fragment, "Chicago," meaning "(a flight connecting in) Chicago (O'Hare airport)." One fragment reply, which must always be interpreted in the context of the system's prompting question, is a confirmation (e.g., "yes") or denial (e.g., "no"). In U(8), the user's "Okay" confirms the booking of flight 1792. In U(9), the user's "No, thank you" denies the price inquiry offered by MERCURY. Finally, in U(10), the user repeats "No, thank you" to deny MERCURY's offer of further assistance.

## 1.3 Outline

The intent of this thesis is to provide the reader with an understanding of the necessity for the new CR server and how it handles context resolution within our current domains. These include JUPITER [53] for weather information retrieval, MERCURY [44] for flight reservations, ORION [41] for online task delegation, PEGASUS [54] for flight arrival and departure information retrieval, and VOYAGER [17] for Boston area traffic and landmark information retrieval, as well as the simpler key-value-based domains created using the SPEECHBUILDER utility [19].

| | |
|---|---|
| U(1): | I need to book a trip from Oklahoma City to Calgary on June 17. |
| M(1): | *Okay, from Oklahoma City to Calgary on Monday, June 17.* *Can you provide the approximate departure time or airline?* |
| U(2): | American, around 1:30 pm. |
| M(2): | *What about a flight connecting in Dallas Fort Worth leaving at 2:08 pm?* *Would that work?* |
| U(3): | What is the arrival time? |
| M(3): | *American flight 3742 arrives at 6:44 pm in Calgary.* *Shall I add this flight to your itinerary?* |
| U(4): | Does it serve dinner? |
| M(4): | *No meal is served on the first leg, and dinner is served on the second leg.* |
| U(5): | Okay, book this flight. |
| M(5): | *American flight 3742 has been added to your itinerary...* *What date will you be returning?* |
| U(6): | Four days later. |
| M(6): | *I have connecting flights in Dallas Forth Worth and Chicago O'Hare on Friday, June 21. Would one of these work?* |
| U(7): | Chicago. |
| M(7): | *There is only one American flight connecting in Chicago O'Hare.* *It leaves at 2:09 and arrives at 9:09 pm. Shall I book this flight?* |
| U(8): | Okay. |
| M(8): | *American flight 1792 has been added to your itinerary...* *Would you like me to find out how much your trip will cost?* |
| U(9): | No, thank you. |
| M(9): | *Okay. Is there something else I can do for you?* |
| U(10): | No, thank you. |
| M(10): | *Thanks for using* MERCURY. *Have a nice day!* |

Figure 1-6: Sample dialogue between a user (U) and MERCURY (M), displaying the resolution of sentence fragments, ellipses, pronouns, definite noun phrases, and temporal references.

The chapters are divided accordingly:

- **Chapter 2: Dialogue, Discourse, and Context Resolution**
  in which we define dialogue and discourse, provide a history of discourse processing in spoken dialogue systems, and place the notion of context resolution within the more general areas of dialogue and discourse.

- **Chapter 3: Improvements to the Framework**
  in which we focus on modifications to the framework supporting context resolution, and how these changes facilitated improvements to the context resolution functionality.

- **Chapter 4: The Context Resolution Algorithm**
  in which each function within the CR algorithm is described in detail.

- **Chapter 5: Evaluation**
  in which we present an evaluation of the new CR server based on a comparison between its performance and the performance of the old CR component.

- **Chapter 6: Summary and Future Work**
  in which we summarize our research on context resolution and suggest possible enhancements for context resolution within the GALAXY domains.

- **Appendix A: Constraint Specification Meta-Language**
  in which we describe how to use the constraint specification meta-language.

# Chapter 2

# Dialogue, Discourse, and Context Resolution

Spoken language interaction between a human and a computer can be as simple as speaking a menu's name, instead of clicking on it, or speaking a command such as "Shutdown" or "Go to sleep." In order to complete a more significant task via a computer, however, such as reserving a theater ticket, a user will likely need to provide multiple, potentially interrelated utterances; in other words, the user will carry on a conversation with the computer. In such a case, the system must maintain some representation of the current conversational state. For example, the machine should remember the utterances that have already been spoken by both conversational participants, as well as be constantly aware of the user's ultimate goal. Such a representation allows the system to interpret each user utterance in the context of the current conversational state.

A spoken dialogue system is able to handle conversational interaction using ideas from the studies of *dialogue* and *discourse*. In general, dialogue and discourse are terms used to describe the meaningful exchange and mutual understanding of utterances between a speaker and one or more listeners. It is extremely challenging for a system to optimally handle such conversational phenomena when only using previously spoken utterances for the interpretive context. These phenomena become increasingly more difficult to handle as the physical context of the user or machine becomes relevant, as it does with gestural or mouse-clicked input.

This chapter will begin with descriptions of dialogue and discourse in terms of a spoken dialogue system. Since context resolution is a subset of discourse processing, as will be explained in Section 2.4, a very brief description of dialogue, and a much more in-depth description of discourse, will be provided. Then, the evolution of discourse processing, as it has been implemented over the years, will be described. Finally, the notion of context resolution, as presented in this thesis, and its role between dialogue and discourse will be explained.

## 2.1 Dialogue

A *dialogue* is an interactive communication between two or more participants [29]. In a spoken dialogue system, the role of the dialogue manager is to decide what should happen next in the conversation. For example, in a hotel reservation dialogue system, if a user has just asked to book a hotel, the system may repeat the booking information (which may, ultimately, be cumbersome for the user to hear) or it may simply say, "Okay, your room has been booked." Judgments must be made as to when the system should confirm, when it should assume, and when it should clarify. Many similar decisions are relevant in the study of dialogue.

A dialogue may be decomposed into a series of *subdialogues*. In a city information spoken dialogue system, for example, a user may initiate a dialogue to obtain directions by asking, "How do I get there from here?" If the system does not know where "there" is, it may branch off from the main dialogic path to initiate a subdialogue in order to obtain the user's intended location. Once the system has this information, it can proceed with the main course of the dialogue. In the context of dialogue systems, researchers strive to formulate algorithms, or *dialogue strategies*, by which a computer and a human may collaboratively participate in a dialogue towards the fulfillment of some task or goal.

## 2.2 Discourse

A discourse may be defined as any form of conversation or talk. This may include a speech, a story, a description, an argument, a negotiation, or a task-oriented dialogue, which is common for a spoken dialogue system.

Research into these various discourse types has resulted in the identification of, and the observation of relationships among, *discourse segments*, the basic units of a discourse. A discourse segment is a group of utterances, which conveys a special meaning, intention, or part of a plan. Grosz [22] notes that:

> The segment-level intention is not a simple function of the utterance-level intentions, however, but a complex function of utterances, domain facts, utterance-level intentions, and inferences about these.

The intention of a discourse segment may be any of proposition, evidence, confirmation, denial, background information, question, or answer, among many others. Thus, *discourse processing* involves interpreting, and identifying the intention of, an individual utterance, grouping such utterances into segments, identifying the intentions of these discourse segments, discovering how these segments relate to one another, and further, identifying any intentions that such interrelations may convey. For example, the interactions among various discourse segments may provide information about a plan that the user is trying to execute.

The need to interpret individual utterances exposes the issues of anaphora, deixis, and ellipsis resolution. *Anaphora* is the term used for a phrase that refers to some previously mentioned entity in the discourse; this may include a pronoun or a definite noun phrase. There have been several approaches to such reference resolution, including Webber's notion of *discourse entities* [47], Discourse Representation Theory [27], and the centering theory of Grosz, Joshi, and Weinstein [21], in which the current focus of attention in the discourse contributes to how the references will be resolved.

*Deixis* can be defined as "the function of pointing or specifying from the perspective of a participant in an act of speech or writing" [51]. Five main types of deixis have been characterized, as described by Loehr in [32]:

- **person**: *I*, *me*, *you*, *we*, ...
- **spatial**: *this*, *that*, *here*, *there*, ...
- **temporal**: *now*, *tomorrow*, *two days later*, ...
- **discourse**: *the former*, *the latter*, *the above*, ...
- **social**: the use of a formal name versus a nickname, the use of familiar versus formal second person pronouns (e.g., Spanish *tú* versus *usted*), ...

Loehr also explores the notion of hypertext linking in webpages as a possible new type of deixis, which he calls *hyperdeixis.* Research in the area of deixis has also included work by Webber [48], for natural language understanding systems, in which she studies the use of deictics to refer to one or more clauses, as opposed to noun phrases. In addition, deixis in spoken language systems has been studied by Eckert and Strube [14].

*Ellipsis* is the phenomenon in which specific information has been omitted from an utterance. This elided information is assumed to be understood and should be retrievable from the previous utterance or from the intentional structure of the discourse [22]. Approaches to ellipsis resolution include those of Hendrix, et al. [25], Carberry [8], and Kehler [28]. Hendrix, et al. obtained the omitted information from the previous user utterance, handling ellipsis by substituting the elliptical phrase for a syntactically analagous phrase in the previous utterance. Carberry was able to obtain the omitted information, not necessarily from the previous utterance, but from the intentional structure of the discourse. Kehler also takes a semantic approach to the problem, utilizing the notion of *role linking* for resolution.

Several theories have arisen that strive to formalize the notion of discourse processing. The main difference among the theories concerns the information deemed relevant for determining the discourse segmentation. The foremost theories include one of discourse coherence by Hobbs[26], a generation-based discourse strategy by McKeown [34], a theory of discourse structure based on attentional state, intentional structure, and linguistic structure by Grosz and Sidner [24], and the Rhetorical Structure Theory of Mann and Thompson [33]. Each one is incomplete, however; hence, research continues to explore and, potentially, to extend the theories.

In short, discourse research deals with two main questions, which are observed by Grosz in [23]:

1. what information is conveyed by groups of utterances that extends beyond the meaning of the individual utterances, and

2. how does context affect the meaning of an individual utterance.

At least one of these questions has been addressed by many researchers in their implementations of dialogue systems over the years. Several examples of these systems will be described in the following section.

## 2.3 Evolution of Discourse Processing

Discourse processing has been actively researched in natural language applications since the early 1970s. Several discourse capabilities have been introduced over the years to create a more natural dialogue between the user and the system. Early text-based systems include those such as LUNAR [50] and Winograd's natural language understanding system [49]. LUNAR was a natural language interface to a database that contained chemical analyses of moon rocks, and resolved pronouns by consulting a list of previously mentioned entities. Winograd's system allowed a user to manipulate a block world by posing English instructions. The system possessed a notion of context and could handle pronoun and some definite noun phrase resolution, as well as ellipsis. The LIFER system [25], a natural language interface for database queries, also featured elliptical input resolution. Other systems such as SAM (Script Applier Mechanism) [12] and GUS [4] focused on discourse processing in limited domains.

In later years, researchers recognized the insufficiency of unimodal input (i.e., text or speech) in dealing with reference, for example, in applications utilizing maps. Consequently, mouse clicks and gesture were combined with natural language systems.

One of the earliest multimodal systems, *Put-That-There* [5], was published in 1980 and combined verbal reference with gestural pointing to manipulate colored shapes. The EDWARD system [6] allows a user to navigate and manipulate a file system via typed natural language and mouse clicks. EDWARD utilizes the *context model* for reference in which each topic is assigned a numerical salience value for subsequent anaphora resolution. EUCALYPTUS [46] is a natural language interface to an air combat simulation system allowing speech input and mouse clicks for reference. The reference resolution component of EUCALYPTUS is focus-based and follows from the work of Grosz and Sidner [24].

Some of the more recent multimodal systems handle discourse processing in various ways. CommandTalk [45, 35], a spoken language interface for battlefield simulations, employs a *Contextual Interpretation* agent within the dialogue manager to resolve pronominal, definite noun phrase, and deictic references by maintaining a salient entity list and focus spaces. Both *situational context* (current state) and *linguistic context* (history) are also used to interpret underspecified commands [13].

TRIPS [15], a collaborative planning assistant developed at the University of Rochester, handles context resolution by updating a *Discourse Context* [2, 7], which contains a salience list, as well as a discourse history. Several other agents and components communicate with the Discourse Context. For example, a reference component utilizes the information to resolve anaphora, ellipsis, and clarification questions. The *Interpretation Manager* and the *Generation Manager* also communicate with the Discourse Context to plan system replies and to display any system updates.

In the WITAS dialogue system [31], a multimodal system for conversations with autonomous mobile robots, the dialogue manager handles discourse processing by maintaining a salience list of entities, as well as a *modality buffer*, which stores mouse clicks until they are bound to some deictic reference.

Some dialogue systems have recently incorporated the rendering of a human face for system output, in addition to the generation of speech [36]. Previous studies have supported the notion that such facial displays actually convey meaningful discourse functions [11], and can be crucial to providing a more natural experience for the user. Other systems render, not only a human face, but a complete embodied human agent for interaction [10]. Rea [9] is such an agent under development at the MIT Media Lab. She is able to sense limited user gestures through cameras and associate discourse functions with those gestures. As more complex systems are developed that can sense the bodily movements and facial displays of a user, these features will have to be incorporated into any new or modified theory of discourse processing.

## 2.4 Context Resolution

Now that the notions of *dialogue* and *discourse* have been introduced, the role of *context resolution* can be described. Discourse processing was previously said to involve the interpretation of individual utterances, the segmentation of the discourse, and the meanings of and interactions among the resulting segments. Context resolution is not concerned with the intentions of discourse segments, but rather, with the interpretation of individual utterances in the immediate or recent context. Specifically, this involves the resolution of anaphora, deixis, and ellipsis, the propagation of "understood" contextual information from earlier utterances, and the resolution of any incompletely specified user requests. In terms of this functionality, context resolution is a subset of discourse processing.

In the GALAXY framework, the dialogue manager (DM) controls the strategy that guides dialogue flow. The DM also handles discourse processing outside of context resolution. Any notion of a user intention or plan, which may span several utterances or the entire dialogue, is addressed by the DM. For example, if a user were to say to MERCURY, "I want to fly to Chicago tomorrow and return two days later," the DM will recognize that the user intends to fly tomorrow *and* he wants a return flight two days later. The DM will know to resolve the initial flight before moving on to resolve the return flight. Even though the DM may encounter several understanding problems while attempting to resolve the initial flight, once it is resolved, the DM will move on to the unresolved user intention—the return flight. Thus, it is a duty of the DM to deal with all current user intentions before it attempts to resolve any additional user intention.

While context resolution is not concerned with the discourse segment intentions or planning, it does make an attempt to reconstruct the intention of a user in the case of a speech recognition error or a fragmented parse. This is a function that is not performed during the context resolution phase in the other dialogue systems mentioned in Section 2.3. We have assigned this functionality to context resolution since any such error requires that the utterance be interpreted in the context of what may be recognized or parsed correctly in the semantic representation. We believe that this issue is consequential of the nature of a spoken dialogue system and is an issue that would need to be addressed in a general theory of discourse processing. Whether or not this intention reconstruction should definitely be a role of the context resolution component, however, is debatable.

## 2.5 Summary

In this chapter, we have introduced the notions of dialogue and discourse in terms of a spoken dialogue system. We presented a more in-depth history of discourse processing in system implementations. Finally, we provided a high-level view of what we call *context resolution* and how it relates to the more familiar disciplines of dialogue and discourse.

In the next chapter, we begin to describe details of the new CR server in terms of its supporting framework and data structures, and how this framework is an improvement on the old CR component.

# Chapter 3

# Improvements to the Framework

The old context resolution component in the GALAXY framework has always been embedded within the natural language (NL) server, whose tasks are numerous and complex. In addition to context resolution, the NL server handles the parsing of a user's utterance via TINA, as well as every aspect of natural language generation, from formatting SQL queries to generating viable replies to a user via GENESIS-II [3].

This embedding of functionality has inhibited a generic and extensible representation for context resolution. After many years of various and hasty modifications to the NL server, CR-related code has manifested itself with high specificity and without consistency. Extensions to CR capabilities cannot be easily or generically implemented and, consequently, essential updates seem to be incorporated randomly for lack of a better protocol. In addition, the functions within the context resolution process are constrained by the specifications and boundaries imposed by the NL server.

The new CR server was developed to encapsulate context resolution functionality. Since the server was created from scratch, we wanted to improve upon as many functions as possible, as well as upon the framework supporting context resolution. This chapter will focus on the latter, highlighting the following improvements to the framework:

1. a network of pointers allowing complete access to the context of a semantic frame via a simple, yet powerful, constraint specification language,

2. a modified history record that contains a list of topics throughout the conversation, as well as the semantic representation for the previous utterance,

3. an external means by which to specify, and easily modify, the sequence of function execution in the CR algorithm, and

4. a notion of *semantic satisfaction* in which a parsed semantic frame is reorganized into its most satisfied state.

Before describing our improvements to the framework in detail, we will give a brief explanation of knowledge representation in the GALAXY domains. This will provide sufficient background information, allowing one to see the inherent benefits of the improvements made to the framework.

## 3.1 GALAXY's Knowledge Representation

In the GALAXY domains (JUPITER [53] for weather information, MERCURY [44] for flight reservations, ORION [41] for online task delegation, PEGASUS [54] for flight arrival and departure information, VOYAGER [17] for Boston area traffic and landmark information, and many others), the linguistic knowledge from an utterance is encoded in a hierarchical semantic frame containing objects identified as one of a small set of predefined types, namely, **clause**, **topic**, **predicate**, or **keyword**. TINA, the parsing component of the NL server, parses from a word graph produced by the speech recognizer and encodes the best hypothesis in a semantic frame representation. This frame is then passed to the CR server for context resolution processing. Figure 3-1 shows a possible semantic frame for the utterance, "I want to fly from Cleveland to San Diego tomorrow."

Notice that the top-level frame in the figure is a clause, *statement*, which contains the *flight* topic, since *flight* is the topic of the main clause in the utterance. Within

```
:clause {c statement
           :topic {q flight
                     :pred {p source
                              :topic {q city
                                        :name "cleveland" } }
                     :pred {p destination
                              :topic {q city
                                        :name "san diego" } }
                     :pred {p month_date
                              :topic {q date
                                        :name "tomorrow" } } } }
```

Figure 3-1: Semantic frame for "I want to fly from Cleveland to San Diego tomorrow."

this topic frame, there are three predicates—*source*, *destination*, and *month_date*—which serve to modify the *flight* topic. Each of these predicate frames contains its own topic frame, and each of these topics is made specific by the value of its own `:name` keyword.

An utterance usually contains only one clause, which represents the type of utterance (e.g., *statement*, *wh_query*, *truth*, *what_about*). Each clause frame may contain a single topic object, which typically represents a noun phrase. An object frame may contain multiple predicates, which often represent adjectival or adverbial modifiers, as well as verb phrases.

## 3.2 Semantic Frame Context

The CR server depends heavily on the ability to traverse the semantic frame to locate **clauses**, **topics**, **predicates**, and **keywords**, as well as the contexts in which they occur. The old CR component had very limited, and overly specific, capabilities for accessing this information. When designing the new CR server, we acknowledged this weakness and realized that a generic and much more powerful mechanism would be advantageous. Thus, the CR server utilizes a strategy of organizing the contents of a frame along the dimensions of **clause**, **topic**, **predicate**, and **keyword**, and providing a structure of pointers that link among them.

The old CR component lacked any sort of language with which to specify the various constraints utilized by its functions. Therefore, it was felt that the new CR component would benefit from a general, but expressive, language in which a developer may simply and intuitively specify such constraints. The new CR server employs such a language, which, when coupled with the new structure of linkage pointers, provides an extremely powerful and useful tool.

In this section, we will demonstrate the shortcomings of the old framework in terms of accessing information in a semantic frame. This will be followed by a detailed description of the new mechanism within the CR server that is used to access the complete context of a frame.

### 3.2.1 The Old Framework

Given a semantic frame hierarchy, the old CR component would access a specific element deep within the nest by traversing several internal frames. Consider the semantic frame given in Figure 3-2.

```
{c intention
   :topic {q flight
            :pred {p destination
                     :topic {q city
                               :name "san diego" } }
            :pred {p month_date
                     :topic {q date
                               :name "tomorrow" } } }
```

Figure 3-2: Semantic frame for “I want to fly to San Diego tomorrow.”

In one instance, we may want to know if the topic of the semantic frame is `flight`. The old CR component would handle this by creating a function, which would simply check the first level of the hierarchy for the desired element. In another instance, we may not only want to know if the topic of the semantic frame is `flight`, but also if that topic contains a `destination` predicate and, further, if that predicate contains a `city` topic. This would require another function to be written.

It would become even more troublesome if we also wanted to know a value at a higher or equivalent level in the hierarchy (i.e., a parent or sibling frame). The old CR component accomplishes this in an overly specific and inelegant manner by storing the desired value. For example, if we know, in advance, that we will have to know the name of each predicate's parent frame, we can store the parent frame name as `:parent`, as shown in Figure 3-3.[1]

```
{c intention
   :topic {q flight
            :pred {p destination
                     :parent "flight"
                     :topic {q city
                              :name "san diego" } }
            :pred {p month_date
                     :parent "flight"
                     :topic {q date
                              :name "tomorrow" } } }
```

Figure 3-3: Semantic frame showing how the old CR component stores a frame's parent name as the `:parent` keyword within that frame.

This technique would become unwieldy if it were to be extended to additionally store the siblings, grandparent, clause, etc., for each predicate. In addition, this information may also not be required for every context resolution function, in which case, it would be unnecessarily occupying space. This method simply does not demonstrate the generality or extensibility which we seek in a context resolution component.

### 3.2.2 The New Framework

We have implemented a generic mechanism, in the new CR server, whereby, within a given semantic frame, every component frame has access to every other frame and to all the contents therein. This is accomplished using a network of nodes which mimics the structure of the semantic frame via pointers. Each object type (i.e., **clause**,

[1]The GALAXY frame libraries do not provide any convenient tools for tracing an object's ancestry within a frame.

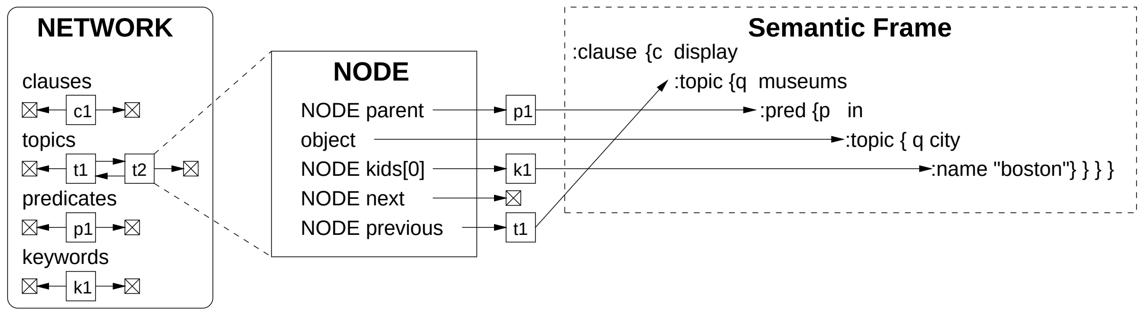


Figure 3-4: Diagram showing how the network of nodes provides access to the complete context of the semantic frame for "Show me museums in Boston."

**topic**, **predicate**, **keyword**) is stored in a separate doubly-linked list to allow easy scanning of all the topics, for example, within the semantic frame. Each list node contains a pointer to its corresponding object in the semantic frame, a pointer to the parent node, and an array of pointers to its children nodes. The parent and children objects may be accessed through the corresponding parent and children nodes. Using this network of pointers, we can access any location in the semantic frame from any location in the semantic frame.

Figure 3-4 shows a diagram of how the network of nodes provides access to the complete context of the semantic frame. The NETWORK structure holds four doubly-linked lists. The *previous* pointer of the first node in each list points to "null" as does the *next* pointer of the last node in each list. In this example, there is one clause (*display*), two topics (*museums* and *city*), one predicate (*in*), and one keyword (*:name*). We will describe here the details of NODE t2. The *parent* member points to NODE p1, whose *object* member points to the *in* predicate frame object in the semantic frame. The *object* member of NODE t2 points to the *city* topic frame object in the semantic frame. The first element of the *kids* array points to NODE k1, whose *object* member points to the *:name* keyword string object in the semantic frame. The *next* member of NODE t2 points to "null" since it is the last topic in the list. Finally, the *previous* member of NODE t2 points to NODE t1, which points to the *museums* topic frame object in the semantic frame. In this case, if our current context is the *:name* object node, we can easily find the nearest encompassing clause, for example, by following the parent pointers until we reach a node that points to a clause object.

The order of nodes in the lists is currently determined by a depth-first traversal of the semantic frame. However, each node also contains a *depth* member, which allows an entire list to be ordered by depth, if so desired.

### Constraint Specification

We have just described the data structures designed to provide complete access to the context of a semantic frame. However, there must also be a means by which one may easily specify the desired constraints to be tested. We have developed a simple meta-language in which a developer may specify the constraints in a linear string format. Appendix A contains a complete specification for this meta-language.

We will now demonstrate several example constraints using the semantic frame in Figure 3-5 as the frame to be tested. We will refer to this frame as "the test frame." Consider the following simple constraint:

```
:topic = connection
```

Since `:topic` is a frame object in the test frame, this constraint means that the name of the frame must be `connection`. If `:topic` were a string object, the constraint would mean that the value of the string must be `connection`. In general, if the object denoted by `:<key>` is a frame, the right-hand side of an expression is expected to be the frame name. If a string object is denoted by `:<key>`, the right-hand side of an expression is expected to be a string. This works similarly for integer and float objects.

Specifying predicate values is a special case. Predicates always occur as frames, never as strings or numeric values. For example, in this constraint:

```
:pred = arrival_time
```

the value in the expression will always be interpreted as a predicate frame name.

In the case where no value is provided with a `:<key>`, the constraint indicates existence. Consider the following:

```
:quantifier & :pred
```

This constraint indicates that the `:quantifier` and `:pred` keywords should simply exist with any value. Since a single frame can have multiple predicates, `:pred` ac-

```
{c truth
  :topic {q connection
           :quantifier "indef"
           :pred {p for
                    :topic {q flight
                              :quantifier "def"
                              :pred {p airline
                                       :topic {q airline_name
                                                 :name "united" } }
                              :pred {p arrival_time
                                       :topic {q time
                                                 :military 1200 } } } } }
```

Figure 3-5: Semantic frame for "Is there a connection for the United flight arriving at noon?" This frame is used to demonstrate constraint testing.

tually means one or more predicates. This constraint also demonstrates the use of the conjunctive operator, `&`, which means that both `:quantifier` *and* `:pred` must exist in the frame for the constraint to be satisfied. Table 3.1 shows some additional simple constraints and their meanings to give an idea of the conditions that can be specified.

The simple constraints are extremely useful, but they do not take full advantage of the benefits provided by the networked node structure. We can make more complex constraints by utilizing the hierarchy operator ( –> ) to indicate moving either up or down in the semantic frame hierarchy. Consider this example in which we move down the hierarchy:

```
:pred = airline  –> (:topic = airline_name  –> :name = united)
```

| Simple Constraint | Meaning |
|---|---|
| `!:quantifier` | `:quantifier` does not exist in the frame |
| `:quantifier = %def` | `:quantifier`'s value contains the substring *def* |
| `:pred != airline` | no predicate exists that has the value *airline* |
| `:name = american \| united` | the value of `:name` is *american* or *united* |
| `:military <= 1200` | the value of `:military` is less than or equal to *1200* |

Table 3.1: Some simple constraints along with their respective meanings.

This constraint matches if there is a predicate with the value, *airline*, and within this predicate frame, `:topic` exists with the value, *airline_name*, and within this topic frame, the keyword, `:name`, exists with the value, *united.* We can see that this constraint is satisfied in the test frame.

Similarly, we can also access information at a higher level in the hierarchy. Consider the following example:

```
    :topic = flight  –> (:P_PARENT = for  –> :T_PARENT = connection)
```

This constraint indicates that the value of `:topic` is *flight*, *flight*'s parent (`:P_PARENT`) is the predicate, *for*, and *for*'s parent (`:T_PARENT`) is the topic, *connection.* This constraint is also satisfied in the test frame. The `:X_PARENT` keys not only allow us to check the parent of a frame, but the type of the frame as well. The above constraint would not be satisfied in the test frame if, for example, *flight*'s parent were not a predicate frame or if *for*'s parent were not a topic frame. The meta-language also supports a generic `:PARENT` key, which satisfies any type.

In some cases, we may want to know what the clause name is, but we do not want to trace through the entire ancestry to find it. Therefore, we also have a mechanism to easily test the clause name as demonstrated by this constraint:

```
:topic = flight  –> :CLAUSE = truth
```

This indicates that the nearest encompassing clause frame of the *flight* topic should have the name, *truth.* Again, this condition holds in the test frame.

### Testing the Constraints

We have described how a developer may specify simple and complex constraints using the new meta-language. When the new CR server is started, all of the constraints present in the external specifications file are converted from their string format into an easily traversable frame representation and are put into storage for subsequent reference.

This means of constraint storage is worthy of mention since it is another improvement on the old CR component by the new CR server. The old component stored all of its constraints in arrays, which was awkward, especially for constraints with multi-

```
{c topic
   :op "="
   :value {c airline_name }
   :name {c name
            :op "="
            :value {c |
                       :arg1 {c american }
                       :arg2 {c %
                                :arg1 {c united } } } } }
```

Figure 3-6: The corresponding condition frame representation for the constraint, `:topic = airline_name  –> :name = american | %united`.

ple arguments. The new CR server utilizes the existing GALAXY frame structures and utilities for constraint storage, as well as for knowledge representation, eliminating the need for additional storage and access functions that must be created for novel data structures.

Figure 3-6 shows the corresponding *condition frame* for the following constraint:

`:topic = airline_name  –> :name = american | %united`

The resulting condition frame can then be sent as an argument to the boolean function, `test_condition(NODE_to_test, condition_frame)`. The first argument is a NODE, which points to the frame to be tested. The NODE structure is passed so that we may access parent or sibling relationships, if they happen to be specified in a constraint.

The function traverses the condition frame and the frame to be tested, verifying each sub-expression. Once a sub-expression evaluates to *false*, the traversal ceases and the function returns *false*. If all sub-expressions in the condition frame are verified, the function returns *true*.

## 3.3 The History Record

The main task of the CR server is to interpret a user's utterance in the context of what has previously been spoken in the dialogue. Therefore, the information stored in history, and how this information is represented, has a significant effect on what information is accessible by the context resolution functions.

The old CR component remembers the context-resolved frame from the previous utterance, which is typically a clause. In addition, the component maintains a series of semantic slots, each of which holds a single object of the given semantic type, such as *source*, *destination*, or *date*. These semantic slots are consulted for the purpose of filling in any missing information in an utterance and for determining the object to which a pronoun may refer, otherwise known as an *antecedent*. This mechanism severely limits the amount of history remembered, which restricts how far back in history the user can refer. A suitable history record should allow the user to make references to objects introduced into the dialogue several utterances back.

Another issue to consider is the extent to which objects should be *processed* before adding them to the history record. One option is to store every semantic frame in a list, and simply traverse this list searching for an appropriate antecedent, for example. This strategy would likely be inefficient in both memory and time, and would place the burden of complex search on the antecedent resolution function.

An alternative strategy is to extract entities from the semantic frame, which may subsequently be referenced, and store them in some convenient structure in the history record. For example, all the topics introduced into the dialogue could be maintained in a list structure. This list could then be traversed to find the most recent instance of a specific semantic type or to find a valid antecedent. This is the strategy adopted in the design of the new CR server.

The new CR server, like the old CR component, remembers the context-resolved frame from the previous utterance. This frame may serve as the context in which a spoken fragment, such as "Tomorrow," would be interpreted. In addition, the server maintains a bounded list of all topics that are introduced into the dialogue. By default, every topic frame is placed into this entity list; however, the domain developer has the power to exclude specific topics by specifying them in an external file. The developer also has control over the number of topics that are maintained in this list. When the maximum number of topics has been enlisted, the oldest topics are displaced to prevent an unrealistically long-term memory.

The topics for this list are extracted from a hierarchical semantic frame in a depth-first manner, so that the topmost entity in the list is the *highest-level* topic from the *most recent* utterance. It is possible for the list to contain multiple entities that match a given search constraint, in which case, the first match will be selected for the task at hand.

The list just described is extensible and can be used to maintain any object, not only topics. In future versions of the CR server, we may discover the need to maintain lists of specific predicates or clauses, in which case this utility can easily be extended.

## 3.4 External Control for Function Sequence

For any system under development, the ability to easily add, modify, or remove specific functions from a procedure facilitates experimentation with various function sequences; additionally, it eases the debugging process. This is not the case in a system in which the function sequence is "hard-wired" into the code. In this situation, in order to disable a specific function, or to modify the existing sequence of functions, the code has to be modified and then recompiled. Thus, the ideal control for function sequence needs to exist outside the code.

The old CR component specifies its function sequence in the code and requires re-compilation after every change. Context resolution, however, is a rather experimental process, frequently requiring modification. Therefore, a context resolution algorithm would benefit a great deal from an external means of function sequence control. This is the strategy adopted by the new CR server.

The CR server utilizes an external *dialogue control table* to specify the order in which its component functions be executed. This same mechanism has previously been used successfully in the GALAXY dialogue managers, the most notable of which is for MERCURY [44]. The developer can specify the desired order of functions in this table along with optional keys which, when present in the current dialogue state, will trigger the execution of a specific function.

```
:*discourse --> check_for_user_clicks
:*discourse --> check_for_one_list_item
:*one_list_item & !:*user_clicks --> record_single_item_as_click
```

Figure 3-7: Excerpt from the CR server's dialogue control table.

Figure 3-7 shows an excerpt from the CR server's dialogue control table. This sequence of rules handles *implicit focus*. In the web-based version of GALAXY, the user may ask for a list of "neighborhoods in Newville." The user could then click on a neighborhood that appeared in a displayed list and follow with, "How do I get *here* from Boston?" The system will link *here* with the clicked neighborhood. If the displayed list, however, showed only a single neighborhood, that neighborhood would be implicitly in focus, as if it had been clicked by the user; but, no explicit user click would be required to correctly interpret the follow-up query.

The keys to the left of a "`-->`" can be set by the functions to the right of the "`-->`" in any of the preceding rules. For example, if `check_for_user_clicks` were triggered, it would create the `:*user_clicks` key in the dialogue state only if the user had, in fact, input a mouse click to the system. If `check_for_one_list_item` were then triggered, and it only found one item in the displayed list, it would add the `:*one_list_item` key to the dialogue state. Then, according to the third rule in Figure 3-7, if `:*user_clicks` did *not* exist in the dialogue state, the next rule would trigger `record_single_item_as_click`, and so on through the sequence of rules.

## 3.5 Seeking the Most Satisfied Semantic Frame

The old CR component did not do any reorganization within the semantic frame that it received from the NL server. Instead, it only handled the incorporation of information from the history into the parsed semantic frame. In other words, the old CR component considered the parser to be entirely correct in its placement of parsed concepts within the semantic frame.

In contrast, the functionality of the new CR server is based on manipulating the semantic frame in order to obtain the most satisfied state possible. The satisfaction state of the semantic frame is determined by the satisfaction of its constituent objects (i.e., **clause**, **topic**, **predicate**, **keyword**). Our system allows the domain developer to specify a set of satisfaction constraints that indicate which objects are licensed to occur within other object frames. Throughout the context resolution process, various unsatisfied objects may be moved around the semantic frame until they attain a state of satisfaction. This rearrangement is typically performed to deal with a *robust parse*, when the semantic frame may contain several, as yet, unrelated fragments. A robust parse results when the parser cannot obtain a single full parse solution, and backs off to parsing and combining fragments [39].

The network structure, described in Section 3.2.2, allows us to access the complete context of any object within a semantic frame. Therefore, the satisfaction of a single object may be dependent on any combination of the other objects within the semantic frame. We now present some examples which will serve to make this concept clearer.

Figure 3-8 shows the semantic frame for the utterance, "Is there a connection for the United flight arriving at noon?" In order to check the satisfaction of the individual objects within the frame, we reference the satisfaction constraints in the external file. Some example constraints, as they may be specified by the developer, are shown in Figure 3-9.

In the figure, the `#<TYPE> = <name>` heading indicates the type and name of an object frame. Under the heading, one may specify any or all of `.satisfied_topics`, `.satisfied_predicates`, and `.satisfied_keys`. Under one of these subheadings come the constraints which the **topic**, **predicate**, or **key** must uphold in order to be satisfied. For example, the first condition specifies that, under a clause frame named *truth*, any `:topic` object will be satisfied. The second condition says that, under a topic frame named *flight*, the following predicates are satisfied:

1. *airline* if it contains a `:topic` named *airline_name* and this `:topic` frame contains the `:name` key with the value, *united*,
2. *arrival_time* if the name of its nearest encompassing clause contains the substring *truth* or is *statement*.

```
{c truth
  :topic {q connection
           :quantifier "indef"
           :pred {p for
                   :topic {q flight
                            :quantifier "def"
                            :pred {p airline
                                    :topic {q airline_name
                                             :name "united" } }
                            :pred {p arrival_time
                                    :topic {q time
                                             :military 1200 } } } } }
```

Figure 3-8: Semantic frame for “Is there a connection for the United flight arriving at noon?”

```
#CLAUSE = truth
.satisfied_topics
:topic

#TOPIC = flight
.satisfied_predicates
:pred = airline  –> (:topic = airline_name  –> :name = united)
:pred = arrival_time  –> :CLAUSE = %truth | statement

#TOPIC = time
.satisfied_keys
:military  –> (:PARENT  –> :PARENT = arrival_time | departure_time)
```

Figure 3-9: Externally specified satisfaction constraints. All of these constraints are satisfied in the frame shown in Figure 3-8.

The third condition says that, under a topic frame named *time*, the key, `:military` is satisfied if the name of its grandparent frame is *arrival_time* or *departure_time*. All of these constraints are satisfied in the semantic frame, shown in Figure 3-8.

This chapter has focused on those structural aspects of the new CR server that are significant improvements over the old CR component. We have discussed the new framework and meta-language for the CR server, which support much more powerful satisfaction constraints utilizing the entire context of the semantic frame. We have discussed the history record and the information that is stored within. We drew attention to the new mode of external sequence control used by the CR server and the benefits thereof. All of these improvements work together to allow very generic and powerful context resolution functionality, which is the topic of the next chapter.

# Chapter 4

# The Context Resolution Algorithm

In the preceding chapters, we have explained the idea of context resolution and how it is applicable to a spoken dialogue system. We have also described why an independent server, in the GALAXY framework, was necessary for a more powerful CR component to be realized. In this chapter, we set out to describe each specific function within the context resolution process.

Determining the exact functions to be carried out by the CR server is a major challenge. The specific functions in the CR algorithm are based on those utilized in the old CR component, in which they were experimentally determined based on collected dialogues [42]. Through the course of developing the new CR server, some functions have been added; some have been deleted; nearly all have been significantly modified. As described in Section 3.4, the algorithm is externally controlled via a dialogue control table; thus, the sequential order of function execution can be easily modified. However, the current configuration of the algorithm, shown in Figure 4-1, has been experimentally determined to provide the best performance.

Before providing a detailed description of each context resolution function, we will give a high-level explanation of the entire CR algorithm.

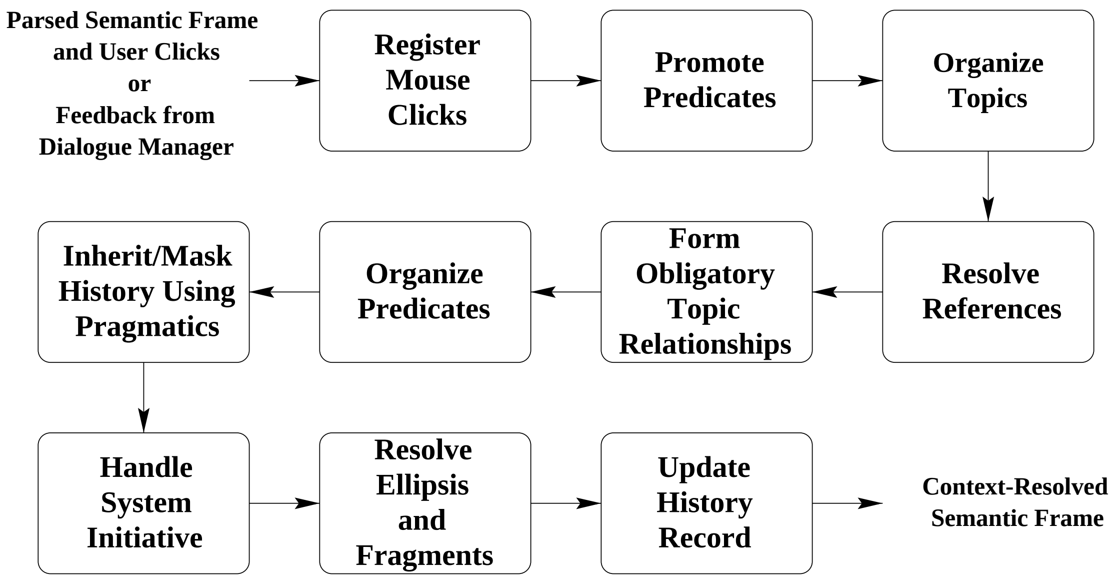

Parsed Semantic Frame and User Clicks or Feedback from Dialogue Manager
Register Mouse Clicks
Promote Predicates
Organize Topics
Resolve References
Form Obligatory Topic Relationships
Organize Predicates
Inherit/Mask History Using Pragmatics
Handle System Initiative
Resolve Ellipsis and Fragments
Update History Record
Context-Resolved Semantic Frame

Figure 4-1: This diagram shows the current configuration of the context resolution algorithm. The sequential order of execution can be modified via an external dialogue control table.

## 4.1 High-Level Description

The process begins when the CR server receives a semantic frame from the NL server. This semantic frame represents the knowledge from the best hypothesis of what the user has spoken.

In addition to speech and typed text, GALAXY's web-based interface allows a user to make references via mouse clicks. The CR server is able to handle references in any of these modalities. Speech and typed text are handled similarly since speech has been transformed to text by the time it reaches the CR server. A mouse click is treated just as if the user had spoken the reference, except that it is made to be more salient than any spoken reference. These clicked references will, ultimately, need to be linked to a verbalized reference such as "here" or "this one." This is performed later in the reference resolution stage.

The semantic frame then passes through a "Promote Predicates" stage that serves to simplify the frame by collapsing nested predicates, with the intent of reducing the depth of the hierarchy. This stage makes the semantic frame shallower and easier to traverse. The promotion of predicates may result in the rearrangement of objects within the semantic frame. The semantic network of pointers, which is newly featured in the CR server, and which was described in Section 3.2.2, must, therefore, be updated to remain consistent with the current and history semantic frames. This update stage is necessary throughout the CR algorithm, whenever the current or history semantic frame is modified, to ensure the consistency of the networks.

As mentioned in Section 1.1.3, one of the roles of the new CR server is to attempt to reconstruct the intention of the user in the case of a robust parse or recognition error. The "Organize Topics" stage partially addresses this issue by finding relationships for unsatisfied **topics** in the current semantic frame. The satisfaction of these **topics** contributes to an increase in the overall satisfaction of the current semantic frame, which is a goal of the CR server.

Reference resolution is an extremely vital function in the CR algorithm. When a user says, "it," "there," or "those flights," or when a user clicks on a map location, it is the job of the "Resolve References" stage to identify the object to which the user is referring. The CR server maintains a *discourse entity list*, or a list of entities throughout the dialogue to which a user may potentially refer. When a reference must be resolved, this list is scanned for an *antecedent* corresponding to the given reference. Reference resolution is difficult due to a potentially high degree of ambiguity, which is simple for a human to process, but is extremely challenging for a spoken dialogue system.

In the next stage, the server establishes obligatory topic relationships as specified in the external file. If one of these relationships does not already exist in the current semantic frame, the CR server will hallucinate it. For example, in the MERCURY domain, if a user were to say, "I want a first-class fare," he actually means "I want a first-class fare (for a flight)." Therefore, *fare for flight* would be an obligatory topic relationship; whenever *fare* exists, *flight* must also exist.

Following the previous stage, the CR server continues to reconstruct a user's intention in the "Organize Predicates" stage by moving unsatisfied **predicates** around the current semantic frame until as many as possible are satisfied. This, again, contributes to the CR server's goal of increasing the overall satisfaction of the current semantic frame.

The propagation of information from the dialogue history is crucial in interpreting subsequent user utterances. The "Inherit and Mask History" stage *inherits* specific information from the history record into the current semantic frame. Some information, which would otherwise be inherited, is *masked* due to the presence of other information already in the current semantic frame. Complex constraints for inheritance and masking are specified in the external file.

Interpreting sentence fragments is easy in human-human communication since we are often unconsciously aware of the context in which they make sense. A dialogue system, however, must be explicitly provided with the proper context for interpretation. One way the dialogue manager handles this is via a *system initiative*. When the dialogue manager prompts the user for a *destination*, for example, it sends a *system initiative frame* to the CR server, anticipating that the user might respond with a fragment such as a city name. If the user were to respond with "Boston," the CR server would use the system initiative frame to interpret "Boston" as a destination, rather than as a source, for instance.

Even after it has been resolved through a system initiative, however, an input fragment may still be a fragment, in which case it needs further processing by the "Resolve Ellipsis and Fragments" stage. This stage resolves phrases from which a user has omitted information. The missing information is assumed by the speaker to be understood by the listener, or in this case, the dialogue system. This phenomenon is prevalent when the user is responding to a system question. However, a fragment, commonly preceded by a "what about" cue, can also occur in the absence of a prompt from the system. Consider the query, "What is the address of John Jones?," followed by "What about Jane Smith?" A fragment is typically resolved by incorporating it into the semantic frame of the previous user utterance.

The final stage in the CR algorithm involves updating the history record. A copy of the newly context-resolved semantic frame, which represents the most immediate historical context, is stored in the history record. It is from this frame that subsequent user utterances will inherit information. The discourse entity list is also updated with every topic frame from the semantic frame.

The context-resolved semantic frame is then sent to the appropriate domain's dialogue manager. If the dialogue manager chooses to modify or update any information in the semantic frame, it will do so and pass the frame back to the CR server to move through the process again. This step is known as a *discourse update*. The algorithm is executed just as if the frame represented the knowledge from a user's utterance. It is also during this stage that a system initiative may be sent to the CR server for subsequent use.

The context resolution functions will now be described in the same sequence in which they are executed in the current configuration of the CR algorithm. For each function, we will describe the functionality as it exists in the new CR server, often providing specific examples. In many cases, we will also draw attention to the ways in which the new CR server improves upon the capabilities of the old CR component.

## 4.2 Register Mouse Clicks

In WebGALAXY[30], GALAXY's web-based graphical interface, a user may point and click on any list item or map location and accompany it with a verbal reference such as "this one" or "there." The first step in the CR algorithm is to incorporate any clicked items as potential *discourse entities* [1], to which the user may later refer. The clicked items are added to this discourse entity (DE) list in temporal order, with the earliest click at the top. Thus, clicked and uttered entities, alike, are put into the same list for subsequent reference resolution. The details of how such references are resolved will be explained in Section 4.6.

The old CR component also supported the resolution of clicks and, in fact, served as the basis for this capability in the new CR server. However, whereas the old CR component only enabled the user to make a single clicked reference per utterance (i.e., any further click would override the previous), the new CR server is able to handle an arbitrary number of clicks simply by retrieving the clicks into an array. However, the CR server obtains a user's clicks from the hub, which stores the click indices as separate variables in the semantic frame used to communicate between the hub and the servers. The number of click variables is currently set at two, allowing a maximum of two clicks per utterance. It is possible to allow additional clicks by setting up a new hub variable for each one, but there is, presently, no simple mechanism supporting an arbitrary number. However, once the hub is able to easily adjust the quantity of user clicks, the CR server will be equipped to handle them.

Although small, the increase from one to two clicks gives a user more reference options, especially in the VOYAGER domain. When presented with a map, for example, a user may ask:

> "Show me directions from MIT to Harvard."

The user, however, might just as easily say:

> "Show me directions from here to there."

while clicking on a starting point (MIT), and on an ending point (Harvard). Such a request can easily be handled by the new CR server, whereas, previously, it could not. At the present, there does not seem to be any good motivation for allowing more than two clicks.

A user is only able to utilize mouse clicks from a graphical display. Therefore, when a user is accessing a GALAXY domain solely via telephone, this function of registering mouse clicks does not apply to, and has no effect on, the context resolution process.

## 4.3 Promote Predicates

The promotion of predicates is a function that was taken directly from the old CR component. Its utility lies in simplifying, or collapsing, nested predicates within a semantic frame. It is debatable whether this stage is truly the responsibility of the CR server, since it could easily be a post-parsing process performed by TINA. Nevertheless, it is currently an obligation of the CR server and will be described here.

Consider the semantic frame in Figure 4-2, in which predicate promotion has not occurred. There are two sets of nested predicates which result from the parser: *depart-when-at* and *arrive-when-around*. Predicate promotion serves to collapse each

```
{c statement
  :topic {q flight
           :pred {p depart
                    :pred {p when
                             :pred {p at
                                      :topic {q time
                                                :military 1200 } } } }
           :pred {p arrive
                    :pred {p when
                             :pred {p around
                                      :topic {q time
                                                :minutes 0
                                                :hour 5
                                                :xm "pm" } } } } } }
```

Figure 4-2: Semantic frame for "I want a flight departing at noon and arriving around five pm" before predicate promotion.

of these nests, bottom-up, into a single predicate for the sake of simplification. The manner in which these nests are collapsed is specified in the external file, in the form of constraints as shown in Figure 4-3. The first condition says that, when the predicate, *when*, is present and either predicate, *at* or *around*, occurs within *when*'s predicate frame, the nested predicates will be collapsed into a single predicate, *at*. Similarly, the predicate-predicate nests, *depart-when* and *arrive-when*, will collapse into the single predicates, *departure_time* and *arrival_time*, respectively, as specified in the second and third conditions. Figure 4-4 shows the much simpler and concise semantic frame resulting from these predicate promotions.

## 4.4 Update Networks and Determine Satisfaction

Now that the semantic frame has been simplified by predicate promotion, we can traverse the semantic frame and create a network of nodes as discussed in Section 3.2.2. This will enable us to utilize context-dependent constraints in any further context resolution processing. Once the network has been established for the semantic frame, the satisfaction of each object within the network is determined. We also establish a

```
#PREDICATE = when
.promotions
at −− > at
around −− > at

#PREDICATE = depart
.promotions
when −− > departure_time

#PREDICATE = arrive
.promotions
when −− > arrival_time
```

Figure 4-3: Externally specified predicate promotion constraints for *when*, *depart*, and *arrive*.

```
{c statement
   :topic {q flight
            :pred {p departure_time
                     :topic {q time
                               :military 1200 } }
            :pred {p arrival_time
                     :topic {q time
                               :minutes 0
                               :hour 5
                               :xm "pm" } } } }
```

Figure 4-4: Semantic frame for "I want a flight departing at noon and arriving around five pm" after predicate promotion.

network of nodes for the history frame, which is the context-resolved semantic frame from the previous utterance, so that our mechanism of history propagation can also be subject to context-dependent constraints.

As the semantic frame is reorganized throughout the context resolution process, its structure changes, rendering the network obsolete. Therefore, after any reorganization of the semantic frame, or the history frame, an update of the corresponding network must be executed.

## 4.5 Organize Topics

As mentioned earlier, we want the semantic frame to move toward as satisfied a state as possible, as it passes through the context resolution process. The first step in establishing this satisfied state is to make sure that the greatest number of topics is satisfied.

The overwhelming majority of topics that result from the parser are already satisfied. In the case of an unsatisfied topic, it is possible that the appropriate licensing constraint simply does not exist in the external file (see Section 3.5). The most likely reason, however, is that the parser was not able to fully parse the user's utterance, resulting in one or more topics for which no relationship could be established. In the old CR component, the relationships between any topics found in an utterance were established simultaneously with the parse of that utterance. In a case where a topic was parsed, but no relationship could be found for it, the topic was simply thrown away, as if it were never mentioned.

Consider the following user utterance:

"I want a hotel in Detroit"

and the corresponding hypothesis from the speech recognizer:

"I want a hotel and Detroit."

in which "and" was substituted for "in." Given that the user did, in fact, mention *Detroit* and *hotel*, we would expect these two concepts to appear as related topics in the parsed semantic frame. Due to the misrecognition, however, the parser could

```
{c identify
   :topic {q hotel
             :quantifier "indef" } }
```

Figure 4-5: Old parser's semantic frame for a robust parse of "I want a hotel and Detroit." The topic, *detroit*, has been discarded.

```
{c identify
   :topic {q hotel
             :quantifier "indef"
             :and_topic {q city
                           :name "detroit" } } }
```

Figure 4-6: Modified parser's semantic frame for a robust parse of "I want a hotel and Detroit." The "extraneous" topic, *detroit*, is retained as an `:and_topic`.

not establish any structure between the topics. Consequently, the second topic was discarded and the semantic frame, as shown in Figure 4-5, was passed to the CR component.

We have implemented a new mechanism in the TINA framework, whereby "extraneous" topics are retained in an `:and_topic` chain in the semantic frame. This was accomplished via a simple modification to one of TINA's control files. The modified parse for "I want a hotel and Detroit," is shown in Figure 4-6.

Under the assumption that the user intended some relevance by uttering every topic, we believe that retaining all the topics will enable the CR server to, potentially, reconstruct the user's intention.

A generic `:and_topic` chain is shown in Figure 4-7. Every `:and_topic` is unsatisfied and we must try to rearrange each one to be in a satisfied state. Our method for satisfying an unsatisfied topic is to find another topic within the semantic frame with which the unsatisfied topic can form a valid relationship. These valid relationships are specified by the developer in the external file. We search the valid relationships to find the appropriate missing predicate, which will then serve to link the two topics.

```
{c clause
   :topic {q topic1
             :and_topic {q topic2
                           :and_topic {q topic3 } } } }
```

Figure 4-7: Semantic frame containing a generic `:and_topic` chain.

```
{c clause
   :topic {q topic1
             :pred {p relation
                      :topic {q topic2
                                :and_topic {q topic3 } } } } }
```

Figure 4-8: Semantic frame in which the `:and_topic`, *topic2* was resolved by the formation of the topic relationship, *topic1 relation topic2*.

### 4.5.1 Forward Topic Relationships

Given the semantic frame in Figure 4-7, we traverse the list of topics from the semantic frame network to identify any possible topic relationships. We check each topic against every topic following it in the list, until a relationship is found. For example, we look at *topic1* and *topic2*. Since *topic2* is unsatisfied, we must find a relationship in which it would be satisfied. Perhaps *topic1 relation topic2* is such a relationship. We consult the externally specified topic relationships table to see if *topic1 relation topic2* exists. If it does, we create the predicate relationship, *relation*, reorganizing the frame to appear as in Figure 4-8. Notice that the `:and_topic` keyword has been changed to `:topic`, so that it may be marked as satisfied. If *topic1 relation topic2* does not exist, we move on, checking if the relationship, *topic1 relation topic3*, is valid. Notice that if *topic2* were already satisfied, there would be no need to check for a valid relationship and we would move on to check *topic1* against *topic3*. Once we are finished with *topic1*, we move on to check *topic2* against every following topic, and so on.

We now present an example. Figure 4-9 shows the parsed semantic frame containing an `:and_topic`, Figure 4-10 shows the relevant topic relationship as specified in the external file, and Figure 4-11 shows the resulting frame after the formation of the forward topic relationship, *fare for flight*.

### 4.5.2 Backward Topic Relationships

The formation of backward topic relationships differs from its forward counterpart because of the way in which the frame must be reorganized. The process is similar, however, in that we check a topic against every topic following it in the list, but we search for the reverse relationship between the two topics. From our example frame in Figure 4-7, we look at *topic1* and *topic2*. Since *topic2* is unsatisfied, we consult the topic relationships table to see if *topic2 relation topic1*, the reverse relationship, exists. If it does, we will create the predicate relationship, *relation*, to obtain the semantic frame as shown in Figure 4-12. If it does not, we move on to check if the relationship, *topic3 relation topic1*, is valid. Notice again, that if both topics were already satisfied, we need not check for a relationship and we would continue.

If a backward topic relationship is created, we update the network's topic list and restart the process. The cycle will end when no more relationships can be formed.

We now present an example. Figure 4-13 shows the parsed semantic frame for "Detroit a rental car," which contains an `:and_topic`, Figure 4-14 shows the relevant topic relationship as specified in the external file, and Figure 4-15 shows the resulting frame after the formation of the backward topic relationship, *rental_car in city*.

This process of topic reorganization provides as satisfied a base as possible, on which the remaining functions can create an even more satisfied semantic frame. To be sure that the semantic frame network is still consistent after this reorganization, we perform an update following this stage in the context resolution algorithm.

## 4.6 Resolve References

The problem of complete reference resolution in human-computer interaction is extremely challenging, if not impossible. However, the inability of a dialogue system to handle references such as "she," "them," "here," "this town," and "that museum" can result in a rather unnatural and awkward dialogue. The ease and quickness with

```
{c statement
   :topic {q fare
            :and_topic {q flight}
            :quantifier "indef"
            :pred {p fare_class
                     :topic "coach" } } }
```

Figure 4-9: Semantic frame resulting from a robust parse of “I need a coach fare uh flight.”

```
#TOPIC_RELATIONSHIPS
fare for flight
```

Figure 4-10: Externally specified forward topic relationship, *fare for flight*, formed to result in the frame of Figure 4-11.

```
{c statement
   :topic {q fare
            :pred {p for
                     :topic {q flight } }
            :quantifier "indef"
            :pred {p fare_class
                     :topic "coach" } } }
```

Figure 4-11: Semantic frame resulting from the formation of the forward topic relationship *fare for flight*.

```
{c clause
   :topic {q topic2
             :pred {p relation
                      :topic {q topic1}
             :and_topic {q topic3 } } } }
```

Figure 4-12: Semantic frame in which the `:and_topic`, *topic2*, was resolved by the formation of the topic relationship, *topic2 relation topic1*.

which a human is able to comprehend and disambiguate such references is remarkable.

The CR server attempts to resolve two general types of references: *anaphoric* and *deictic*. An anaphoric reference is any reference that refers to something previously mentioned in the dialogue. This includes pronouns and definite noun phrases. A deictic reference can be described as verbal pointing. Given the context of a map, for example, a user may refer to a location by saying, "here" or "there." Similarly, when presented with a list of flights, a user may refer to a specific flight by saying, "this" or "that one."

In this section, we describe how the new CR server handles the resolution of such references. The resolution algorithm relies on the maintenance of a discourse entity (DE) list. In general, a DE list contains previously mentioned entities in the conversation, to which a dialogue participant may later refer. An entity can be as simple as a name such as "John," a complex noun phrase such as "the flight from Boston to San Francisco leaving sometime in the morning connecting in Chicago and arriving sometime in the early evening," or even an action situation such as "my going home and your not liking it." Any object, action, idea, situation, or state can be considered an entity. The CR server, currently, considers an entity to be any topic frame object in the semantic frame following the complete context resolution process.

The mechanism of utilizing a DE list is rather simple. Given a reference, the DE list is linearly scanned to find an appropriate entity to which the reference is being made, also known as the *antecedent*. A DE list can be ordered in several ways. The entities may be appended to the list according to recency. This means the first entity to be checked in a search would be the most recently occurring entity. Other systems

```
{c identify
   :topic {q city
             :name "detroit"
             :and_topic {q rental_car
                           :quantifier "indef" } } }
```

Figure 4-13: Modified parser's semantic frame for "Detroit a rental car."

```
#TOPIC_RELATIONSHIPS
also< rental_car in city
```

Figure 4-14: Externally specified backward topic relationship, *rental_car in city*, formed to result in the frame of Figure 4-15. The `also<` tag indicates that the relationship is licensed both as a forward and as a backward topic relationship.

```
{c identify
   :topic {q rental_car
             :pred {p in
                     :topic {q city
                              :name "detroit" } }
             :quantifier "indef" } }
```

Figure 4-15: Semantic frame for "Detroit a rental car" resulting from the formation of the backward topic relationship *rental_car in city*.

rank the entities in order of salience. Centering theory, as described in [1], proposes a list of *potential next centers* ordered by grammatical role. The most salient role is subject, followed by object, indirect object, and any other discourse entities. Consider the following sequence of sentences:

1. The boys gave Mary an apple.
2. She thanked them and ate it.

Figure 4-16 shows the corresponding DE list ordered by recency, following sentence 1. Each entry contains the entity name, along with gender and number features.

| *top of DE list* | |
|---|---|
| `entity:` | **an apple** |
| `gender:` | *neuter* |
| `number:` | *singular* |
| `entity:` | **mary** |
| `gender:` | *feminine* |
| `number:` | *singular* |
| `entity:` | **the boys** |
| `gender:` | *masculine* |
| `number:` | *plural* |

Figure 4-16: A simple DE list ordered by recency for "The boys gave Mary an apple."

When "she" is encountered in sentence 2, the DE list must be searched to find an antecedent that matches the constraints imposed by "she," namely, `gender:` *feminine* and `number:` *singular*. The first entity that matches these constraints is **mary**. The search for "them" must find an entity with `number:` *plural*; the `gender` does not matter since there is only one third person plural pronoun. The first and only match found is **the boys** with `gender:` *masculine* and `number:` *plural*. Finally, the search for "it" only matches on **an apple** with `gender:` *neuter* and `number:` *singular*. Having resolved all the pronominal references, we can present the meaning as:

| | | | | | |
|---|---|---|---|---|---|
| *She* | *thanked* | *them* | *and* | *ate* | *it* |
| **Mary** | *thanked* | **the boys** | *and* | *ate* | **(the) apple** |

This example demonstrates the simple process of using a DE list to resolve references. The same method is utilized by the CR server; the structures are slightly different, however. For example, each entity is a topic frame, and all of its features are the keys within that frame. The constraints that each reference imposes on a potential antecedent are specified by the developer in an external file.

The ability to specify constraints on an antecedent is a new feature offered by the CR server. In the old CR component, the handling of matching on antecedent features was very limited and it was "hard-coded," rendering the functionality rather inextensible. To provide an example of these constraints supported by the new CR server, Figure 4-17 shows the encoding of constraints that would be used to resolve the pronouns in "She thanked them and ate it."

```
#REFERENCE = she
.antecedents
:topic = person  −> :gender = feminine & :number = singular

#REFERENCE = them
.antecedents
:topic = person  −> :number = plural

#REFERENCE = it
.antecedents
:topic  −> :gender = neutral & :number = singular
```

Figure 4-17: Externally specified antecedent constraints used to resolve the references in "She thanked them and ate it."

A developer's ability to modify the antecedents for a reference can also support user utterances in which the pragmatic rules of anaphora and antecedents are violated. For example, it is colloquially common, at least in some parts of America, for a speaker to refer to a singular, indefinite person with the third person plural pronouns, "they" and "them." A situation, such as the following, is not uncommon:

"You said that someone left me a message. What did they say?"

In this case, the normally plural pronoun, "they" refers to the singular form, "someone." If a dialogue system's reference resolution component were very strict about matching on the number feature, it may not be able to make sense of this query. In the GALAXY domains, if a developer were to foresee such speech being used in a given domain, an antecedent rule could be added, licensing specific singular entities to be valid antecedents for “they” and “them.”

In addition to pronouns, definite noun phrases are also references that need to be resolved in context. A definite noun phrase is a noun quantified by a definite article (e.g., “the”) or a demonstrative adjective (e.g., “this,” “those”). When a user says, “Show me that museum,” there must be some museum of which the user believes both parties are aware; otherwise, the utterance would be nonsensical. This museum must have been the topic of conversation at some point if the user is able to refer to it as such.

We will now consider the dialogue in Figure 4-18 between a user (U) and VOYAGER (V), our city guide domain. To add entities to the DE list, the CR server makes a depth-first traversal of a semantic frame, adding each topic frame to the list as it is found.

U(1): What museums do you know in Boston?
V(1): *Here is a map and a list of museums in Boston...*

U(2): Give me the Museum of Fine Arts.
V(2): *Here is the Museum of Fine Arts...*

U(3): Now show me the libraries in Cambridge.
V(3): *Here is a map and a list of libraries in Cambridge...*

U(4): Show me that museum again.
V(4): *Here is the Museum of Fine Arts...*

Figure 4-18: Sample dialogue between a user (U) and VOYAGER (V) to demonstrate the use of a discourse entity list.

Figure 4-19 shows the accumulated DE list following utterances U(1)–U(3). None of the noun phrases in U(1)–U(3) needed resolving. In U(1), “museums” is indefinite and, therefore, does not refer to something previously mentioned. In U(2), "the Museum of Fine Arts" is an unambiguous definite entity. In U(3), "the libraries in Cambridge" is a definite noun phrase, but the added information, "in Cambridge," is sufficient to disambiguate this set of libraries from another. In U(4), we encounter a definite noun phrase, "that museum," which requires resolution since no further information is provided in the utterance to help us disambiguate the reference. The modifier, "again," indicates that the entity has probably been mentioned earlier, but this information happens not to be included in the semantic frame.

The resolution protocol is to consult the valid antecedent constraints in the external file and then to search the DE list for a match. The antecedent constraint for "that museum" is shown in Figure 4-20. We then proceed to scan the DE list starting at the top, looking for the first matching entity. The third entity in the list is the only one satisfying our constraint:

```
:topic = museum  –> (:number = singular | !:number)
```

Thus, this topic will replace the topic we were trying to resolve. The selected antecedent is removed from the DE list since it is now part of the semantic frame. When the context resolution algorithm has been completed, the DE list is updated by extracting the topics from the semantic frame. The topic just referenced, even though it may have been mentioned several utterances back, is now promoted to the top of the list, and will receive higher priority as an antecedent in the following utterance.

## 4.7 Form Obligatory Topic Relationships

The next stage in the CR algorithm addresses the issue of a user speaking in elliptical form. The phenomenon of ellipsis can be defined as speech in which a speaker omits specific information, which he assumes to be understood by the listener. For example, when a person asks someone for the time, he might typically say, "Do you have the time?" The listener would most likely understand this request as "Do you have the time (of day)?" rather than "Do you have the time (of month/year)?" The listener makes an assumption (probably unconsciously) about the unspoken relation-

*top of DE list*

```
{q library
   :context 2
   :quantifier "definite"
   :number "pl"
   :pred {p in
            :topic {q town
                      :name "cambridge" } } }
```

```
{q town
   :context 2
   :name "cambridge" }
```

```
{q museum
   :context 1
   :quantifier "definite"
   :name "Museum of Fine Arts" }
```

```
{q museum
   :context 0
   :number "pl"
   :pred {p in
            :topic {q town
                      :name "boston" } } }
```

```
{q town
   :context 0
   :name "boston" }
```

Figure 4-19: The accumulated DE list following user utterances U(1)–U(3) in Figure 4-18. The list is ordered by depth-first topic retrieval.

```
#REFERENCE = :topic = museum  -> :quantifier = definite | demonstrative
.antecedents
:topic = museum  -> (:number = singular | !:number)
```

Figure 4-20: Externally specified antecedent constraints used to resolve the definite noun phrase in "Show me *that museum* again."

ship between "time" and "day." This is the process we are trying to realize in the CR server.

Using the above hypothetical example, if a user were to ask the system for the time, the CR server would receive "time" as the only topic. The server would assume that whenever it saw the topic, "time," it would interpret this as "time of day" and it would physically add the topic, "day," to the semantic frame, creating the relationship, "time of day." The resulting semantic frame might appear as in Figure 4-21.

```
{c identify
   :topic {q time
            :pred {p of
                     :topic {q day } } } }
```

Figure 4-21: Hypothetical semantic frame for "Do you have the time?" The information, "of day," is assumed by the CR server and is added to the parsed frame.

In the CR server, the developer has total control as to which topic relationships should be considered obligatory. Maintaining our goal of domain-independence, these relationships are specified in an external file. We will now demonstrate how an obligatory topic relationship is created.

A prime example of an obligatory topic relationship is *fare for flight* in the MERCURY flight reservation domain. In this domain, a *fare* is always considered to be *for* some *flight*, which means that anytime the *fare* topic occurs, the *flight* topic must also occur. When a user says, "What is the fare?" the corresponding parse frame in Figure 4-22 is sent to the CR server.

```
{c wh_question
   :topic {q fare
            :quantifier "which_def" } }
```

Figure 4-22: Semantic frame for "What is the fare?"

When the obligatory topic relationships stage in the algorithm is encountered, the server scans the semantic frame's topic list and, for each topic, checks if there is an obligatory topic relationship specified in the external file. When the table is checked for *fare*, the obligatory topic relationship *fare FOR flight* is found as shown in Figure 4-23.

```
#TOPIC_RELATIONSHIPS
fare FOR flight
```

Figure 4-23: Externally specified obligatory topic relationship, *fare for flight*. The obligatory feature is encoded in the capitalized relationship, "FOR."

Each obligatory topic relationship is also licensed as a regular topic relationship as described in Section 4.5. The capitalized predicate, `FOR`, indicates that the topic relationship is obligatory.

Once an obligatory topic relationship constraint has been identified, the semantic frame is checked to see if the topic relationship already exists. The relationship, "fare for flight," would already exist if, for example, the user had explicitly spoken both topics, as in "What is the fare for the flight?" If an obligatory topic relationship already exists, nothing needs to be done. However, if it does not exist, we create, or hallucinate, the required predicate and topic to satisfy the relationship. In doing this, the CR server is assuming that the user simply omitted the information *for the flight* from the utterance knowing, whether consciously or not, that this information was understood by the system. In order to form this relationship, we create a predicate frame named *for*, which contains a topic frame named *flight*. This frame is shown in Figure 4-24.

```
{p for
   :topic {q flight } }
```

Figure 4-24: Hallucinated predicate and topic frames to satisfy the obligatory topic relationship, *fare for flight*.

This predicate frame is then added to the topic frame in Figure 4-22. Figure 4-25 shows the semantic frame following this addition to result in the relationship, *fare for flight*.

```
{c wh_question
   :topic {q fare
            :quantifier "which_def"
            :pred {p for
                     :topic {q flight } } } }
```

Figure 4-25: Semantic frame resulting from the formation of the obligatory topic relationship *fare for flight*.

This stage of forming obligatory topic relationships takes place following reference resolution because some compulsory relationships may be satisfied by the resolution of a specific reference. The external function sequence control in the new CR server is very useful in this situation, allowing the order of the reference resolution and obligatory topic relationships functionality to be easily modified and the results, thereof, to be examined. We will now present an example of this situation.

If a user were to say, "What is its aircraft?," with the intention that "its" be the possessive pronoun for some flight, the semantic frame, shown in Figure 4-26, would be sent to the CR server.

```
{c wh_question
   :topic {q aircraft
            :pred {p for
                     :topic {q pronoun
                              :name "it" } } } }
```

Figure 4-26: Semantic frame for "What is its aircraft?"

Assume that the obligatory topic relationship, *aircraft for flight*, exists. If the formation of obligatory topic relationships took place before reference resolution, the CR server would hallucinate *for flight* since it does not already exist in the semantic frame. The *pronoun* would likely be resolved to *flight* in the following stage, thus, satisfying the obligatory topic relationship, *aircraft for flight*, making the CR server's hallucination in the previous stage unnecessary and semantically incorrect. If the references are resolved first, however, the *pronoun* in this example will likely be resolved to *flight*, satisfying the obligatory topic relationship, *aircraft for flight*. Now, when the obligatory topic relationships are checked, the CR server will not need to hallucinate *aircraft for flight*.

This stage only deals with elliptical fragments, from which the user has omitted topics. It will not handle such ellipses as "What about (flights arriving at) four thirty?" and "What about (flight) four thirty? (Is it also a non-stop flight?)" in which there are multiple interpretations of the utterance given the context of the dialogue. The handling of this type of ellipsis will be described later in Section 4.11.

Since this stage in the CR algorithm may result in modification of the semantic frame, the network of pointers must be updated before proceeding to the next stage.

## 4.8 Organize Predicates

This stage is similar to the topic reorganization stage described in Section 4.5; however, this stage deals with the reorganization of predicates to put the semantic frame in the most satisfied state possible. This stage in the new CR server deals with reorganizing the semantic frame and, thus, has no equivalent in the old CR component. It is an improvement, nonetheless, since an unsatisfied predicate resulting from a robust parse can potentially be moved to a location in the semantic frame where it is satisfied.

As explained in Section 3.5, each predicate is satisfied in the semantic frame wherever the developer licenses it. It is most likely that the developer will specify these constraints to match where the parser places the predicates; however, the developer does have the power to specify how the CR server should rearrange the predicates in a parsed semantic frame, according to the satisfaction constraints in the external file.

One example of this concerns the utterance, "I want a first-class fare from Atlanta to St. Louis." In other words, the speaker wants a first-class fare *for a flight* from

Atlanta to St. Louis. The parser will place *source* and *destination* under the *fare* topic, as shown in Figure 4-27, because there is no *flight* topic present, but a fare does not really have a source or destination—a flight does.

```
{c statement
   :topic {q fare
             :pred {p source
                      :topic {q city
                                :name "atlanta" } }
             :pred {p destination
                      :topic {q city
                                :name "saint louis" } } } }
```

Figure 4-27: Semantic frame for "I want the fare from Atlanta to Saint Louis?"

This is reflected by the satisfaction constraint in the external file, as shown in Figure 4-28. This constraint says that *source* and *destination* predicates are satisfied under the *flight* topic. This constraint is not specified for the *fare* topic, so *source* and *destination* are unsatisfied under the *fare* topic.

```
#TOPIC = flight
.satisfied_predicates
:pred = source
:pred = destination
```

Figure 4-28: Externally specified predicates that are satisfied under the *flight* topic.

Therefore, if there were a *flight* topic, *source* and *destination* could be moved from the *fare* topic to the *flight* topic. The CR server uses its obligatory topic relationships function to hallucinate the *flight* topic. Then, during this stage of organizing predicates, the entire list of predicates is scanned and the satisfaction of each is assessed. Finding *source* and *destination* to be unsatisfied under the *fare* topic, the server will move them under the *flight* topic. The resulting frame will appear as shown in Figure 4-29.

```
{c statement
  :topic {q fare
         :pred {p for
                :topic {q flight
                        :pred {p source
                               :topic {q city
                                       :name "atlanta" } }
                        :pred {p destination
                               :topic {q city
                                       :name "saint louis" } } } } } }
```

Figure 4-29: Semantic frame for "I want the fare from Atlanta to Saint Louis?" following the hallucination of *for flight* and the movement of *source* and *destination* from the *fare* topic to the *flight* topic.

Since the semantic frame is reorganized during this stage, the network of node pointers must be updated so it remains consistent with the semantic frame.

## 4.9 Inherit and Mask History

The next stage is a very important one in the process of context resolution in any conversation. Consider the dialogue in Figure 4-30 between a customer (C) and a travel agent (A), in which the customer is trying to book travel between the mainland and an island off the coast.

Throughout this dialogue, A is acquiring and remembering travel-related details. For example, before A(5), A has gathered and remembered the name of the island (Good Weather Island), the dates of departure (July 1) and return (July 5), the mode of travel (airplane), and the airport of departure (Metro). In C(5), when C tells A that he no longer wants to take the plane, A must decide what information to remember and what information to forget. Since the only change is to the travel mode, A should remember information that is independent of travel mode, and reconfirm or forget all information that is dependent on the travel mode. In A(6), we can see that A has chosen to remember the travel mode, the name of the island, and the dates. Notice that the *plane* travel mode has been overridden by the *boat* travel mode, and that

A(1): What island are you traveling to?
C(1): Good Weather Island.

A(2): What date are you leaving for Good Weather Island?
C(2): July first and I'm coming back on the fifth.

A(3): Would you like to take a plane or a boat to the island?
C(3): I'll take a plane.

A(4): Which airport do you want to leave from?
C(4): How about Metro Airport.

A(5): Okay, you want to take a plane from Metro Airport to Good Weather Island on July first and you want to return on July fifth.
That will cost five hundred dollars.
C(5): On second thought, I'll take a boat.

A(6): Okay, you want to take a boat to Good Weather Island on July first and you want to return on July fifth.
Which port do you want to depart from?
C(6): Port Louise.

Figure 4-30: Sample dialogue between a customer (C) and a travel agent (A), displaying inheritance and masking phenomena.

*airport* is no longer relevant since a boat departs from a port. The *boat* travel mode has made the travel agent forget the *airport* detail of the travel.

This stage in the CR algorithm deals with the issues just described in the sample dialogue. The CR server must determine what information from the dialogue history should be incorporated (inherited) into the current context for the user's utterance, and what information should be forgotten or overridden (masked). This is accomplished via domain-dependent inheritance and masking constraints, which are specified by the developer in an external file, thus, maintaining the domain-independence of the CR server code.

The old CR component handles the inheritance and masking of information from history, but its power is restricted by its limited constraint specification. The meta-language designed for the new CR server, described in Section 3.2.2, allows much more detailed constraints and significantly increases the power of inheritance and masking. The relevance of these detailed constraints to each of inheritance and masking will be explained in the corresponding section below.

### 4.9.1 Inheritance

The constraints for inheritance are specified by the developer in an external file. In the above sample dialogue, the equivalent CR server constraints may appear as in Figure 4-31.

```
#TOPIC = travel_plan
.inherited_predicates
:pred = destination
:pred = departure_date
:pred = return_date
:pred = travel_mode
:pred = departure_airport
:pred = departure_port
```

Figure 4-31: Externally specified inheritance constraints for the *travel_plan* topic.

These constraints indicate that, when the topic of the dialogue is *travel_plan*, the following list of "inherited_predicates" are licensed to be propagated from the dialogue history.

The new meta-language supports very specific inheritance constraints. If the developer were to choose to only remember a *destination* predicate if it was an island and it had the name *Good Weather Island*, the following detailed constraint might be specified:

```
:pred = destination  -> (:topic = island  -> :name = Good_Weather_Island)
```

This would mean that any *destination* predicate from the history that did not satisfy this constraint, would not be inherited into the context of the current semantic frame.

A constraint of this specificity is not possible in the old CR component; thus, its inheritance capabilities are limited.

While the old CR component only deals with the inheritance of predicates, the new CR server supports predicate inheritance, keyword inheritance, and even a form of topic inheritance. The new CR server divides its inheritance into three types:

1. inheritance from an antecedent
2. inheritance from the previous utterance
3. inheritance of obligatory information

The first type of inheritance in the CR algorithm is from an antecedent selected during the reference resolution stage. When an antecedent frame is chosen, it is temporarily stored in the history record, and an empty frame, with the antecedent's name, is created in the current semantic frame to replace the reference. Then, during the inheritance stage, predicates and keywords are inherited from the stored antecedent frame into this newly created frame. This type of inheritance is useful for incorporating features of the selected antecedent, which may have been mentioned several utterances back.

The second type of inheritance is from the context-resolved semantic frame of the previous utterance. This frame represents the most immediate historical context of the dialogue, including specific information propagated from earlier utterances. In this type of inheritance, the CR server simply scans the topic list in the current semantic frame and, for each topic, locates a single topic, in the previous utterance's frame, from which it may inherit. If it cannot find a topic with the same name, it backs off to finding a topic of the same semantic class. The inheritance constraints are consulted and, if valid, the information from this history topic is propagated. This same process is repeated for the top-most clause in the current semantic frame, so that both clause and topic frames may inherit information.

The third type of inheritance deals with a form of ellipsis described earlier in Section 4.7. For example, if a user were to simply say, "Show me," perhaps he had previously mentioned a restaurant, a museum, or any landmark that might be displayed; and he assumes that the system will understand what is meant. The utterance, "Show me," results in a *display* clause. The system can then search the history record for a landmark, which is capable of being displayed. This *landmark* is called an *obligatory topic*, since the *display* clause is meaningless without it. If no such *landmark* can be found in the history, nothing is done, and the CR algorithm continues, leaving this issue to be handled by the dialogue manager.

Earlier, in Section 4.7, the *obligatory topic relationship* was described. The obligatory topic relationship is a **topic-pred-topic** construct that must exist in the current semantic frame. If it does not exist, the complete relationship is hallucinated by the CR server. In contrast, an obligatory topic is enforced under a **clause** frame, resulting in a **clause-topic** construct. Also, if an obligatory topic does not exist, it will *not* be hallucinated by the CR server.

In a similar manner, the CR server also supports *obligatory predicates* under **clauses** and **topics**. For example, if a user simply said, "The distance." The developer may choose to specify obligatory predicates for this **clause**, such as *from* and *to*, since a distance must be between two locations. Figure 4-32 shows how such a constraint specification may appear.

```
#CLAUSE = distance
.obligatory_predicates
:pred = from  -> :topic = location | landmark | roadway
:pred = to  -> :topic = location | landmark | roadway
```

Figure 4-32: Externally specified obligatory predicate constraints for the *distance* clause.

There is, of course, a design decision made in implementing these obligatory topics and predicates. If a user were to say, "Show me," it is possible that choosing the most recently mentioned landmark to display may not be what the user intended. An alternative would be to initiate a clarification subdialogue to resolve the ambiguity. The system could, for example, reply, "What landmark would you like me to show you?" Nevertheless, if the system is incorrect in its decision, the user will typically be more explicit in his request and the system will probably be able to capture the user's intention correctly.

### 4.9.2 Masking

It is possible to completely prevent the propagation of specific information simply by omitting it from the inheritance constraints in the external file. In some contexts, however, it is desirable for a specific predicate or keyword to be propagated, while in others, it is best to discard it. The mechanism of *masking* is combined with *inheritance* to handle this situation.

Recall the travel agent/customer dialogue above. The agent wants to propagate the *destination_airport* information, which is relevant since the *travel_mode* is *plane*. When the customer switches *travel_mode* to boat, however, the *destination_airport* is no longer relevant and the agent now wants to forget that information; in other words, the agent wants to mask *destination_airport*. The CR server's constraint for such masking would appear in an external file, as shown in Figure 4-33.

```
#PREDICATE = departure_airport
.masked_by
:travel_mode = boat
```

Figure 4-33: Externally specified masking constraint for the *departure_airport* predicate.

This constraint signifies that a predicate named *departure_airport* in the history will not be incorporated into the parsed semantic frame if the parsed semantic frame contains the keyword, `:travel_mode`, and it has the value, *boat*. The old CR component cannot handle such a masking constraint since there is a value imposed on the predicate; the old constraint specification language is too limited to handle such an issue.

We will now show an example of inheritance and masking in the MERCURY flight reservation domain. Consider the dialogue in Figure 4-34 between a user (U) and MERCURY (M).

U(1): I want to go from Boston to Denver tonight, connecting in Cleveland.
M(1): *Sorry, I could not find any such flights.*

U(2): Is there a nonstop flight?
M(2): *I have three nonstop flights from Boston to Denver tonight...*

Figure 4-34: Sample dialogue between a user (U) and MERCURY (M), displaying the masking of *connection_place* by *flight_mode = "nonstop"*.

In U(1), the user provides source, destination, date, and connection city constraints for a flight. This knowledge is represented in a semantic frame as shown in Figure 4-37. In M(1), MERCURY notifies the user that no such flights could be found. In U(2), the user modifies his constraints, requesting a nonstop flight instead of a connecting one. The knowledge from U(2) is represented in the semantic frame shown in Figure 4-38.

The inheritance procedure begins by scanning the topic list of the current semantic frame, which, in this case, is the frame in Figure 4-38. The topic list is obtained via a depth-first traversal, resulting in *flight* and *non-stop*. The first topic, *flight*, is considered. Looking at the history frame, in this case Figure 4-37, we find the topic from which *flight* is most likely to inherit. An exact topic name match and exact parent name match is the most preferred, followed by only an exact topic name match, and then by a topic name with the same semantic class as *flight*. We find a matching topic name, so the *flight* topic from history is the topic from which we will inherit.

There are four predicates in this topic frame: *source*, *destination*, *month_date*, and *connection_place*. The inheritance constraints table is consulted and all predicates are found to be inheritable by the topic, *flight*, as shown in Figure 4-35.

```
#TOPIC = flight
.inherited_predicates
:pred = source
:pred = destination
:pred = month_date
:pred = connection_place
```

Figure 4-35: Externally specified predicates that the *flight* topic may inherit.

We first consider the *source*, *destination*, and *month_date* predicates. Since none of these predicates already exists in the current semantic frame, each one may potentially be inherited. We check each one, in turn, to see if it will be masked by some other predicate that already exists in the current semantic frame. After consulting the masking constraints table, no maskers are found for *source*, *destination*, or *month_date* and all three predicates are inherited.

```
#PREDICATE = connection_place
.masked_by
:pred = flight_mode  -> :topic = nonstop
```

Figure 4-36: Externally specified masking constraints for the *connection_place* predicate.

We finally consider the *connection_place* predicate. It does not already exist in the current semantic frame, so we move on to check for maskers. After consulting the masking constraints table, we find that a *flight_mode* predicate with a "nonstop" topic in the current semantic frame will mask a *connection_place* predicate from the history. This is specified in the masking constraints table as shown in Figure 4-36. This makes sense since the presence of both predicates, *flight_mode = nonstop* and *connection_place*, in a semantic frame would incur conflict. By masking *connection_place*, the user's most recent intention receives the focus, and the system understands, "I want a nonstop flight tonight from Boston to Denver," in which *source*, *destination*, and *month_date* are inherited and *connection_place* is masked. Figure 4-39 shows the resolved semantic frame following inheritance and masking.

### 4.9.3 Pragmatic Verification

The inheritance mechanism utilized by the old CR component had previously inherited, for example, predicates from the history without acknowledging the validity of the relationship between the inheriting topic and any topic of the predicate being inherited. Thus, the old CR component recently began using a utility that queries a database to verify the pragmatics of an inheritance before it actually occurs. This utility is currently only used in the JUPITER weather domain, but it has proven to be successful for its intended purpose.

A developer can specify relationships that need pragmatic verification in the external file. For example, a *city* and *state* combination needs to make sense; i.e., the combination, "San Francisco, Vermont," is invalid, as is the *city* and *country* combination, "Bratislava, England." Thus, when an inheritance will result in *city in state* or *city in country*, pragmatic verification is required. Figure 4-40 shows how these constraints are specified in the external file.

We will now present an example of how this pragmatic verification functions. Consider the dialogue in Figure 4-41 between a user (U) and JUPITER (J), the weather information system.

The user is trying to find out the weather in various cities. In U(1), the user contributes the city, "Chicago," and the state, "Illinois," to the dialogue. There is no dialogue history, so there can be no inheritance to pragmatically verify. In U(2), the user mentions the city, "Springfield." This overrides the city, "Chicago," from the history. Next, the server determines whether "Illinois" should be inherited. A database pragmatics check on *city = Springfield* and *state = Illinois* verifies the *city-state* combination and the inheritance is allowed.

This utterance also illustrates one technique that JUPITER uses to handle ambiguous city names. The city, "Springfield," exists in many states, including Illinois,

```
{c intention
   :topic {q flight
            :pred {p source
                     :topic {q city
                               :name "boston" } }
            :pred {p destination
                     :topic {q city
                               :name "denver" } }
            :pred {p month_date
                     :topic {q date
                               :name "tonight" } }
            :pred {p connection_place
                     :topic {q city
                               :name "cleveland" } } } }
```

Figure 4-37: Semantic frame for "I want to go from Boston to Denver tonight, connecting in Cleveland."

```
{c truth
   :topic {q flight
            :pred {p flight_mode
                     :topic "nonstop" } } }
```

Figure 4-38: Parsed semantic frame for "Is there a nonstop flight?" This utterance follows that in Figure 4-37.

```
{c truth
   :topic {q flight
            :pred {p source
                     :topic {q city
                               :name "boston" } }
            :pred {p destination
                     :topic {q city
                               :name "denver" } }
            :pred {p month_date
                     :topic {q date
                               :name "tonight" } }
            :pred {p flight_mode
                     :topic "nonstop" } } }
```

Figure 4-39: Semantic frame for "Is there a nonstop flight?" following inheritance and masking from Figure 4-37.

```
#DATABASE_REQUEST = geography@sql
.verify
city in state
city | state in country
```

Figure 4-40: Externally specified relationships requiring pragmatic verification.

U(1): Can you tell me the weather in Chicago, Illinois?
J(1): *The weather in Chicago in Illinois is...*

U(2): How about in Springfield?
J(2): *The weather in Springfield in Illinois is...*

U(3): How about in Detroit?
J(3): *The weather in Detroit is...*

Figure 4-41: Sample dialogue between a user (U) and JUPITER (J), displaying pragmatic verification of city and state to allow or to prevent inheritance.

Massachusetts, and Missouri. JUPITER resolves this ambiguity by inheriting the state, "Illinois," from the history record. [1]

In U(3), the user mentions the city, "Detroit." This overrides the city, "Springfield," from the history. Next, the server determines whether "Illinois" should be inherited. A database pragmatics check on *city = Detroit* and *state = Illinois* returns no results, signifying that the pair is invalid; the inheritance will be prevented, and "Illinois" is discarded from the immediate context.

Since inheritance and masking may significantly modify the structure of the current semantic frame, the network for the current semantic frame must be updated following this stage.

[1] If there were no state in the history record to inherit, the dialogue manager would resort to a clarification subdialogue with the user to disambiguate the city.

## 4.10 System Initiatives

In order for a conversational dialogue system to be practical, its components must be configured for a specific domain such as flight reservations, hotel bookings, or theater reservations. This allows the recognizer and grammar of the system to concentrate on a select vocabulary, since an unconstrained lexicon would be huge, and understanding the infinite number of ambiguous utterances would be immensely difficult. For example, the utterance, "I want to go tomorrow," in a domain-generic system could carry, literally, hundreds of meanings. If this same utterance were issued in a theater reservation domain, however, the system would know that the person, most likely, wanted to attend some theater event tomorrow.

Nevertheless, ambiguity does not disappear once the focus is on a single domain. If a user were to say, "April first," in a flight reservation domain, he may intend either to depart or to arrive on April first. However, if the system had just asked the question, "What date will you be returning?," April first could be unambiguously understood as a return date. In addition, responses such as "Yes" and "No" can only be understood in the context of the system's question. For example, "Shall I book this flight?" "*Yes.*"

This phenomenon is handled jointly by the dialogue manager and the CR server via a *system initiative*. In order to know "*what* on April first" or "Yes, *what*" the prompting question issued by the system must be known. The prompts are determined according to the dialogue strategy, which is controlled by the dialogue manager. Before the system prompts a user, it sets up a context frame, or *system initiative frame*, for the subsequent user utterance, anticipating a fragment response. This frame is sent to the CR server, which stores it in the history record. The system then replies to the user with the prompting question. If the user responds with an ambiguous fragment, the CR server checks if the fragment is a valid response to the system initiated question. If it is valid, the CR server has succeeded in disambiguating the fragment, and incorporates it into the current semantic frame or the history semantic frame. (This incorporation into the history semantic frame will be explained in

Section 4.11.) If the user's utterance is not a valid response to the system's prompt, the CR server will attempt to interpret the utterance in a later stage.

In order to clarify this process, we will present an example. Consider the short dialogue in Figure 4-42 between a user (U) and MERCURY (M). The user provides a destination in U(1). According to its dialogue strategy, MERCURY will next prompt the user for a source. The dialogue manager creates a system initiative frame for *source* and sends it to the CR server, where it is stored in the history record. MERCURY then queries the user in M(1). In U(2), the user gives a fragment response, "Boston," which is intended to be the departure city.

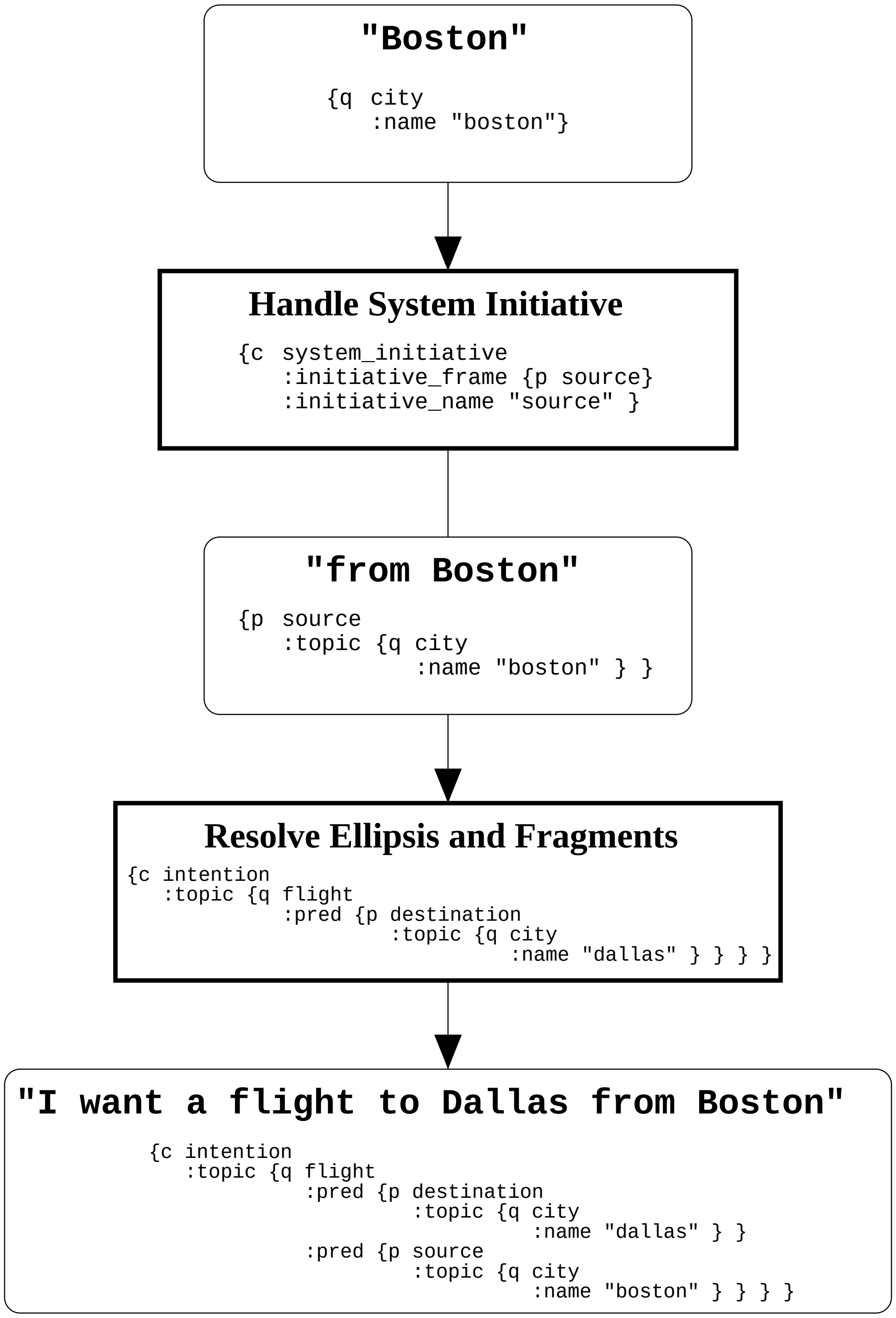
U(1): I want a flight to Dallas.
M(1): *Where does the flight depart from?*

U(2): Boston.

Figure 4-42: Sample dialogue between a user (U) and MERCURY (M) to illustrate how a *system initiative* is used to process an ambiguous fragment response.

Figure 4-43 illustrates how the CR server disambiguates this fragment and incorporates it into the current context. The first box shows the spoken fragment, "Boston," and its corresponding semantic frame, as input to the "Handle System Initiative" function. This function finds the *source* system initiative frame in the history record and it must test if "Boston" is a valid response. The system developer is in control of assigning valid user responses to a given system initiative; they are specified in the external file. Figure 4-44 shows the valid user responses to the *source* system initiative. This constraint indicates that either a *city* or an *airport* is a valid response. Since "Boston" is a *city*, it is a valid response and it will be incorporated into the system initiative frame. The output of "Handle System Initiative" shows the resulting semantic frame, which indicates that "Boston" has been interpreted as a *source*, "from Boston," via the system initiative.

Notice that the system initiative frame in this example is a **predicate**. In such



"Boston"
{q city :name "boston"}
Handle System Initiative
{c system_initiative :initiative_frame {p source} :initiative_name "source" }
"from Boston"
{p source :topic {q city :name "boston" } }
Resolve Ellipsis and Fragments
{c intention :topic {q flight :pred {p destination :topic {q city :name "dallas" } } } }
"I want a flight to Dallas from Boston"
{c intention :topic {q flight :pred {p destination :topic {q city :name "dallas" } } :pred {p source :topic {q city :name "boston" } } } }

Figure 4-43: This diagram shows how a fragment response is disambiguated in the "Handle System Initiative" stage, and is further incorporated into the dialogue context via the "Resolve Ellipsis and Fragments" stage of the CR algorithm.

```
#SYSTEM_INITIATIVE = source
.responses
city
airport
```

Figure 4-44: Externally specified valid responses to the *source* system initiative.

a case, the input fragment may become partially disambiguated, but it will still be a fragment, and it must further be incorporated into the semantic representation. However, it is also possible for a system initiative frame to be a **clause**. In this situation, the transformed input would no longer be a fragment, and it would not require further resolution.

If the output of the "Handle System Initiative" stage requires further processing, this will be accomplished in the "Resolve Ellipsis and Fragments" stage of the CR algorithm. Continuing the above example, this latter stage will produce the final context-resolved intention, "I want a flight to Dallas from Boston," as shown in the figure. The mechanism for this incorporation will be described in Section 4.11.

Again, since this stage may have resulted in the modification of the current semantic frame, as well as the history semantic frame, both the current and history semantic networks need to be updated upon completion of this stage.

## 4.11 Ellipsis and Fragments

Many subtleties contribute to the naturalness of a human-human dialogue, one of which is the ability to communicate using incomplete sentences, or fragments. An ellipsis is an utterance from which the speaker has omitted information, which he assumes to be understood by the listener. Consider a travel agent who is requesting information from a customer to book a flight to Tahiti for him. If the customer says, "I want to leave tomorrow," his intention is that he wants to fly on a plane to Tahiti tomorrow. The customer assumes that "fly on a plane to Tahiti" is understood by the travel agent, eliminating the need to explicitly say, "I want to fly on a plane to Tahiti tomorrow."

A fragment can be thought of as a special case of ellipsis, in which the omitted information is the entire remaining context of the sentence. If the customer says, "Tomorrow," he assumes the travel agent knows that tomorrow is when the customer wants to fly on a plane to Tahiti.

In the case of ellipsis or a fragment, a human listener is, remarkably, able to hear such a construct and interpret it instantly and correctly using, not only, the context of the ongoing dialogue, but also world knowledge, semantic relationships, inference, and common sense.

In a spoken dialogue system, in particular our GALAXY systems, there is limited access to the above-mentioned resources. Therefore, rather simple, but powerful, mechanisms are used to interpret both ellipses and fragments. It is common for systems to interpret ellipses and fragments by incorporating them into the semantic representation of the previous utterance [42, 1, 25]. The CR server accomplishes this in two ways. An ellipsis is typically resolved by inheriting the "understood" information from the history record during the inheritance and masking stage of context resolution. A sentence fragment is commonly resolved by splicing it into the context-resolved semantic frame stored in the history record.

There is a challenge associated with splicing a fragment into the semantic frame of the history; that is, what if there are multiple locations into which the fragment may be spliced? Consider the sample dialogue in Figure 4-45 between a user (U) and VOYAGER (V).

U(1) results in the context-resolved semantic frame, shown in Figure 4-46. This frame is then stored in the history record.

U(2) is an elliptical fragment; the procedure is to splice it into the semantic frame of the previous user utterance. Figure 4-47 shows the parsed semantic frame for U(2).

The resolution protocol is to scan all the topics in the history frame, Figure 4-46 in this example, searching for the most appropriate topic to be replaced by the new fragment. The history topics are prioritized as follows, in terms of which one should

U(1): Show me a neighborhood in Boston.
V(1): *Here are the neighborhoods in Boston...*

U(2): What about Cambridge?
V(2): *Here are the neighborhoods in Cambridge...*

Figure 4-45: Sample dialogue between a user (U) and VOYAGER (V), displaying the use of semantic constraints for ellipsis resolution.

```
{c display
   :topic {q neighborhood
            :pred {p in
                     :topic {q town
                               :name "boston" } } } }
```

Figure 4-46: Semantic frame for "Show me a neighborhood in Boston."

be replaced:

1. a topic that has the same name
2. a topic that has the same semantic class

It is important to note that "neighborhood" and "town" are in the same semantic class of *region*. Therefore, there is ambiguity in terms of the topic to replace. However, the protocol says, if only one topic exists with the same name as the fragment, that topic will be chosen for replacement. This decision overrides the ambiguity created by the presence of several topics with the same semantic class. Nevertheless, if multiple topics exist with the same name, the fragment is ambiguous as to how it should be incorporated into the dialogue context, and no replacement is made. Likewise, if

```
{c what_about
   :topic {q town
            :name "cambridge" } }
```

Figure 4-47: Semantic frame for the elliptical phrase, "What about Cambridge?"

multiple topics with the same semantic class exist, and none has the same name as the fragment, no replacement is made.

In the given example, only one topic exists with the same name as the fragment, i.e., *town*. Therefore, the ambiguity is overridden and *town* is chosen as the topic to replace. The resolved frame is shown in Figure 4-48.

```
{c display
   :topic {q neighborhood
            :pred {p in
                     :topic {q town
                              :name "cambridge" } } } }
```

Figure 4-48: Semantic frame after ellipsis resolution of “What about Cambridge?”

In the case that the fragment cannot be resolved unambiguously, it is passed on to the dialogue manager. The dialogue manager may acknowledge this ambiguity and adjust its dialogue strategy accordingly.

When the new CR server considers which topic to replace in the history semantic frame, it scans the topic list obtained from the semantic network. The old CR component is not able to correctly handle the above sequence of utterances, because it does not consider all the topics in the history semantic frame due to the limited breadth-first search it performs to locate the topics. Consequently, the best topic to be replaced in the history semantic frame may never even be considered, simply due to the incomplete scan of the frame. The semantic network in the new CR server, however, allows easy access to *all* topics in the semantic frame, so the ideal topic to replace may be identified.

An ellipsis or fragment resolution may modify both the current and history semantic frames. Therefore, the corresponding semantic networks need to be updated in order to remain consistent.

## 4.12 Update History Record

The last stage of context resolution involves updating the history record. The history record has been referenced throughout this chapter, without much detail given to how it is maintained. This section will describe the maintenance of the history record.

The history record is a major source from which the CR server obtains the context in which to interpret user utterances. There are two major components to the history record. The first is the discourse entity list, which, potentially, contains every topic mentioned in the conversation. The developer has the power to prevent specific topics from being stored in the entity list; these topics are listed in the external specifications file. The developer may also impose a bound on the length of the entity list. The second component is the context-resolved semantic frame of the previously spoken utterance.

In order to update the discourse entity list, the topic frames from the current semantic frame must be extracted and appended to the existing list, which may already contain other entries from earlier in the dialogue. The topic frames are identified during a depth-first traversal of the current semantic frame; each topic frame is appended to the discourse entity list as it is found. This updated list can then be consulted to resolve references in subsequent utterances.

The final step is to store the current context-resolved semantic frame in the history record, in its entirety. It is from this frame that information is inherited and masked by subsequent utterances. This stored frame also serves as the frame into which a subsequent sentence fragment may be spliced.

## 4.13 Feedback from the Dialogue Manager

When the context resolution process has been completed, the resolved semantic frame is sent to the dialogue manager for the appropriate domain. The dialogue manager may then modify the semantic frame, if necessary, and/or set up a system initiative.

One instance in which the dialogue manager would modify the semantic frame is in the case of a relative temporal reference (e.g., next Friday). The dialogue manager will convert this reference into an absolute date and send the modified semantic frame back to the CR server. The modified semantic frame will proceed through the CR algorithm as if it were the parsed semantic frame representing a user's utterance. The absolute date would then be incorporated into the dialogue history, overriding the relative date spoken by the user.

The dialogue manager may also set up and send a system initiative to the CR server in this stage. The CR server will store the system initiative in the history record and will utilize it in processing the next user utterance.

Heretofore, we have described each function in the CR algorithm in great detail, while providing several examples. This algorithm is used by the major domains under development by SLS, including JUPITER, MERCURY, ORION, PEGASUS, and VOYAGER. These rather complex domains utilize hierarchical frames for knowledge representation. In the next section, we describe how the CR server is being used to support context resolution in simpler domains, typically developed using the SPEECHBUILDER software tool, which utilizes an alternate knowledge representation—a flat list of key-value pairs.

## 4.14 Key-Value-Based Context Resolution

In addition to supporting context resolution in the previously mentioned domains, which utilize hierarchical frames for knowledge representation, the CR server is also being used to support context resolution in domains featuring a simpler key-value (KV) based knowledge representation. This context resolution capability for KV pairs is novel in the CR server; it was not supported by the old CR component.

This functionality was originally designed to be used by SPEECHBUILDER, a web-based software tool allowing the non-expert developer to build a spoken dialogue interface to an application. SPEECHBUILDER uses the existing GALAXY servers for its speech and language technology requirements, but it additionally provides utilities to simplify the creation of a new domain for a non-expert developer. The KV-based knowledge representation utilized by SPEECHBUILDER also facilitates various knowledge and constraint specifications. This KV-based knowledge representation requires KV-based context resolution and it has allowed us to investigate the potential of context resolution based solely on a flat list of keys and values.

SPEECHBUILDER is intended to be novice-friendly and, thus, requires as simple a constraint specification as possible. The context resolution constraints are currently automatically generated as a default. The developer may then easily and quickly modify the constraints to increase the complexity of the domain, if so desired.

While work continues on SPEECHBUILDER, SLS has also made progress on a generic dialogue manager [38], which can handle simple dialogue strategies in any domain. The KV-based context resolution approach has been favored over the hierarchical frame-based version to be used in conjunction with this dialogue manager. While the more complicated hierarchical frames provide much greater power than the KV pairs, the latter are a more intuitive way for the non-expert developer to specify constraints. Simplicity and easy comprehension of constraint specification is a prime goal of SPEECHBUILDER, the generic dialogue manager, and the new CR server.

KV-based context resolution is not yet as complex as that supporting the hierarchical frame representation, but work is continuing to increase the functionality. The KV-based context resolution process consists of inheriting a KV pair from history into the current semantic frame, if the key is not already present in the frame. This inheritance can also be prevented, or masked, by other keys already present in the current frame. The server is also able to determine if a user's utterance is the reply to a system initiative. Each of these phenomena will now be explained.

### 4.14.1 KV-Based Inheritance

The inheritance mechanism is rather simple. The function will scan the list of keys obtained from the network representing the history semantic frame. Each KV pair will potentially be inherited into the current semantic frame. An external table contains information about which keys may be inherited by a specific clause. Consider the dialogue in Figure 4-49.

U(1): What is the phone number for Sam McGillicuddy?
S(1): *The phone number for Sam McGillicuddy is 123-4567.*

U(2): How about Pam O'Grady?
S(2): *The phone number for Pam O'Grady is 765-4321.*

Figure 4-49: Sample dialogue between a user (U) and a faculty info domain (S), displaying inheritance using a KV-based knowledge representation.

The semantic frame for U(1) is shown in Figure 4-50. The frame name, *eform*, is simply the name given to a frame utilizing the KV-based knowledge representation. No information is inherited into this frame since no history record exists yet. Once processed, however, this semantic frame is entered into the history record. When U(2) is spoken, the semantic frame in Figure 4-51 is created by the parser and is sent to the CR server. The server consults the external inheritance table to check which keys may be inherited given that the clause of the current semantic frame is *request*. The relevant inheritance constraints appear in Figure 4-52. Each licensed key will be inherited from the frame in Figure 4-50 into the frame in Figure 4-51 if the key is not already present in the latter frame. Only `:property` will be inherited, resulting in the context-resolved frame, shown in Figure 4-53.

### 4.14.2 KV-Based Masking

The masking mechanism works to prevent the propagation of keys, which would otherwise be inherited, from the history. The presence or absence of specific keys or KV pairs in the current semantic frame may cause other keys to be masked. For example, consider an alternate U(2) in the above dialogue: "How about O'Grady?" The user still wants Pam O'Grady's phone number, but he omitted the first name, since people often refer to a person using only a surname. If inheritance took place, without any masking, the resolved frame would represent, "What is the phone number for Sam O'Grady?" If we impose the constraint that `:first_name` be masked from

```
{c eform
   :clause "request"
   :property "phone number"
   :first_name "Sam"
   :last_name "McGillicuddy"
   :domain "LCSinfo" }
```

Figure 4-50: KV-based semantic frame for "What is the phone number for Sam McGillicuddy."

```
{c eform
   :clause "request"
   :first_name "Pam"
   :last_name "O'Grady"
   :domain "LCSinfo" }
```

Figure 4-51: KV-based semantic frame for "How about Pam O'Grady?"

```
#CLAUSE = request
.inherited_and_satisfied_keys
:property
:first_name
:last_name
```

Figure 4-52: Externally specified inheritable keys for the *request* clause.

```
{c eform
   :clause "request"
   :property "phone number"
   :first_name "Pam"
   :last_name "O'Grady"
   :domain "LCSinfo" }
```

Figure 4-53: KV-based semantic frame for "What is the phone number for Pam O'Grady?" after inheriting the `:property` key.

history if the current semantic frame contains `:last_name`, as shown in Figure 4-54, then the semantic frame in Figure 4-55 will result. This will be interpreted as desired, “What is the phone number for O’Grady?” In the case that there is more than one O’Grady in the database, the dialogue manager may adjust its strategy to handle this ambiguity.

```
#KEY = :first_name
.masked_by
:last_name
```

Figure 4-54: Externally specified masking constraint for the `:first_name` key.

```
{c eform
   :clause "request"
   :property "phone number"
   :last_name "O’Grady"
   :domain "LCSinfo" }
```

Figure 4-55: KV-based semantic frame for “What is the phone number for O’Grady?” after inheriting `:property` and masking `:first_name`.

### 4.14.3 KV-Based System Initiatives

System initiatives were described earlier in Section 4.10. A system initiative is a frame sent to the CR server by the dialogue manager, which sets up the context in which the following user utterance will be interpreted. For example, if a user were to say, “June third,” in a hotel reservation domain, he could be referring to either the check-in date or the check-out date. The system initiative tells the CR server how that date should be interpreted.

This small set of context resolution functions has worked well in the simple domains explored to date. In the future, we want to extend the capabilities of these domains, however, making them more complex and able to handle phenomena such as reference resolution and, perhaps, the reconstruction of a user's intention in the case of a robust parse. Current and future work will focus on researching various algorithms and structures, pursuing the best way to realize such extended functionality given the KV-based nature of this knowledge representation.

## 4.15 Summary

In this chapter, we have described each context resolution function in detail. We have given several examples of how the new CR server resolves a user's utterance in context and, in some cases, how this resolution is an improvement over that offered by the old CR component.

In the next chapter, we will describe the method used to evaluate the CR server. We will also present evaluation results, which demonstrate how the new CR server performed in comparison to the old CR component.

# Chapter 5

# Evaluation

The performance of the CR server is difficult to assess. Nearly every stage comprising the CR algorithm could certainly be evaluated as a single entity. In fact, there are several systems which concentrate solely on what is a single stage in our CR algorithm, most notably, reference resolution. Such a system could be evaluated by running it on some input and having it output a mapping from each reference to its corresponding referent. Then, the output could simply be compared to an accepted mapping of each reference and a percentage of correct resolutions could be obtained.

In evaluating the new CR server, however, we chose to take a holistic approach. The goal of the CR server is not to perfect the performance of any single function; rather, the output of the collective context resolution process is what should be assessed. While each function is made to be very powerful and to perform as optimally as possible, our evaluation centered on the collaboration between these functions to produce a resolved semantic frame most closely reflecting the user's intention.

The purpose of the new CR server is to improve upon and to completely replace the old CR component. In order to achieve this goal, the CR server must successfully handle context resolution in all of the existing GALAXY domains, namely JUPITER, MERCURY, ORION, PEGASUS, and VOYAGER. The old external files containing context resolution constraints had to be translated into the different syntax utilized by the new CR server. This task was completed, and minimal testing was carried out, for all domains. Since MERCURY is the most complex domain currently under development, we chose to perform an extensive evaluation in this flight reservation domain. To do so, the performance of the new CR server was evaluated by comparing its resolved output to that of the old CR component.

As we just mentioned, the old context resolution constraints had to be translated into the new specification. While the power of the new constraint specification is much greater than that of the old one, the equivalent constraints were represented in the new file so that the performance could be compared at a common level.

If we had used the power of the new constraint specification to its full potential, the new CR server would be able to handle situations impossible for the old component. The intention was to verify that the new CR server performed *at least* as well as the old component, and that had to be done using comparable constraints. We did, in fact, find that the CR server performed as well as the old component in most cases, better in several cases, and worse in only a couple of cases. These results will be discussed later. First, we present the details of the evaluation procedure.

## 5.1 Procedure

Evaluation was performed offline on user utterances contained in log files from real user telephone interactions with GALAXY in the MERCURY flight reservation domain. This offline processing could be accomplished by using the *batchmode* server in conjunction with pre-recorded data in *log files*.

### 5.1.1 Batchmode

The purpose of the batchmode server is "to generate inputs to a system during offline processing" [18]. This *batchmode* process is depicted in Figure 5-1. A batchmode run differs from an online run in that the N-best list of hypotheses is obtained from a log file, rather than from the recognizer; the differing portion of an online process is shown dimmed in the figure.

The ability to bypass the audio server and recognizer facilitates the processing and evaluation of, potentially, hundreds of utterances contained in a large number

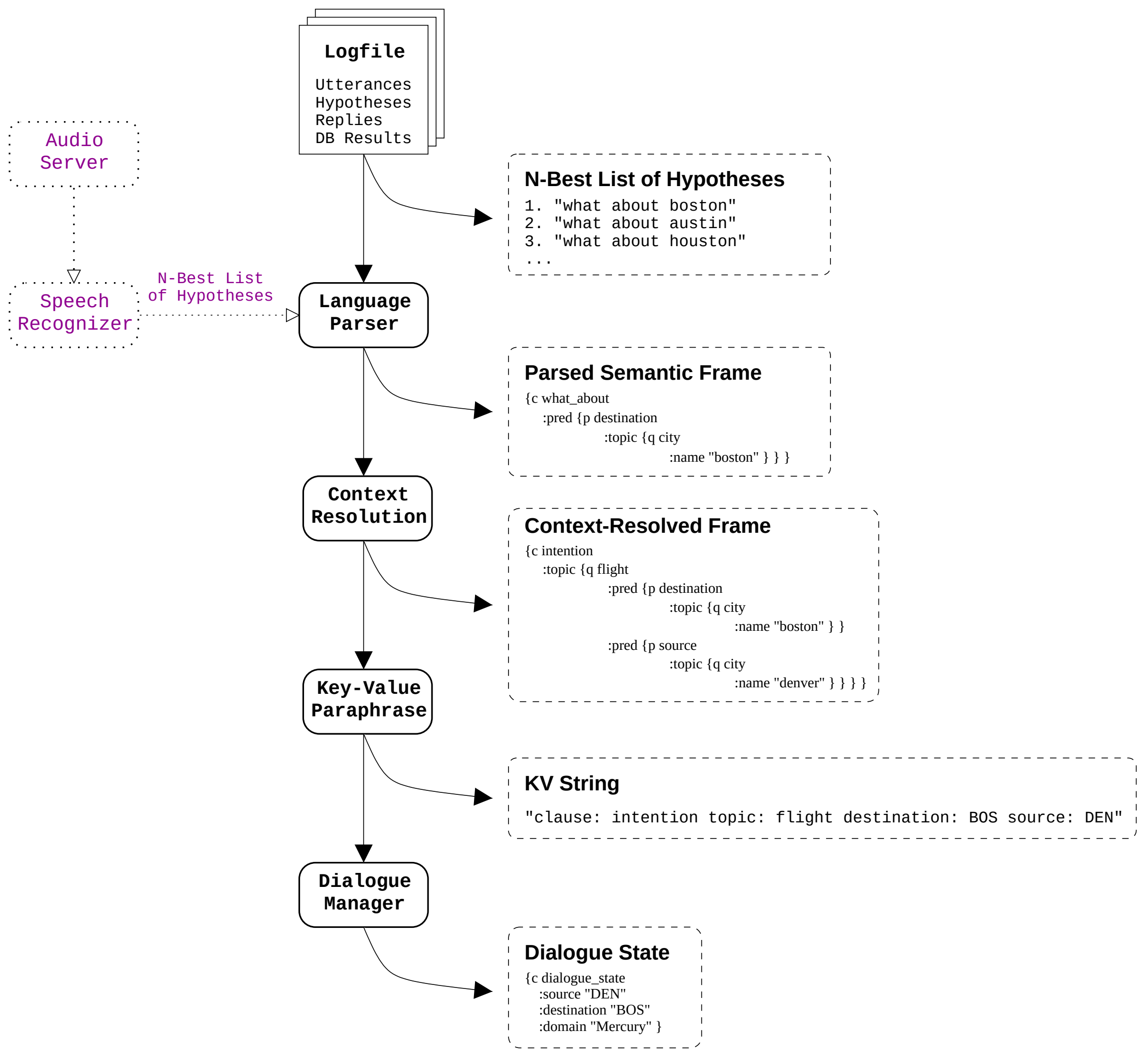

Logfile
Utterances
Hypotheses
Replies
DB Results
Audio Server
Speech Recognizer
N-Best List of Hypotheses
Language Parser
Context Resolution
Key-Value Paraphrase
Dialogue Manager
N-Best List of Hypotheses
1. "what about boston"
2. "what about austin"
3. "what about houston"
...
Parsed Semantic Frame
{c what_about :pred {p destination :topic {q city :name "boston" } } }
Context-Resolved Frame
{c intention :topic {q flight :pred {p destination :topic {q city :name "boston" } } :pred {p source :topic {q city :name "denver" } } } }
KV String
"clause: intention topic: flight destination: BOS source: DEN"
Dialogue State
{c dialogue_state :source "DEN" :destination "BOS" :domain "Mercury" }

Figure 5-1: This diagram shows the system flow during an offline batchmode evaluation, in which the hypotheses are obtained from log files. An online mode would obtain the hypotheses from the speech recognizer; this path is shown dimmed.

of dialogues. In such understanding components of a spoken dialogue system, the recognition performance is not always relevant. This is the case for context resolution evaluation, so we chose to circumvent the audio and recognition servers, obtaining the necessary inputs from log files processed by the batchmode server.

## 5.1.2 Log Files

A *log file* contains information that is generated by the dialogue system as a live interaction is occurring. A developer is able to control the content of a log file by listing specific elements in the *hub program* [43], which guides communication among the various GALAXY servers. A developer may choose, for example, that the system output the parsed semantic frame, the context-resolved semantic frame, or even the results for each database query. Processed dialogues that store the database results in the log file may be evaluated at any future time, without the danger of a dialogue becoming incoherent due to different results obtained from the dynamically updated database.

### 5.1.3 Unit of Evaluation

The unit of evaluation was a key-value (KV) string. Figure 5-1 shows how GENESIS-II produces a KV string for each utterance following the parsing and context resolution stages. A KV string is simply a string representation of the knowledge within a semantic frame. The KV string following context resolution represents the knowledge parsed from the user's utterance as well as any information propagated from the history record. The KV string is then used by the dialogue manager to determine the dialogue state and, consequently, the system's next move in the dialogue.

## 5.2 Experiment

The same dialogue system was run twice. The first system, henceforth known as "the old system," used the old CR component for context resolution, while the second system, henceforth "the new system", used the new CR server for context resolution. The corresponding KV strings from each run were compared. If the two KV strings were identical, the new CR server was counted as having performed equivalently to the old CR component on that utterance. Every remaining pair of KV strings differed in some way. Each difference was placed into one of four categories in terms of how the new system performed on the corresponding utterance: *neutral*, *better*, *worse*, or *excluded*.

A difference was designated *neutral*, or insignificant, if the old and new systems resulted in the same dialogue state. A consequence of this is that the replies to the user after the given utterance would be the same in both systems. A difference in the KV strings could result in the same dialogue state if, for example, the information passed to the dialogue manager by the CR server were overridden due to the longer-term knowledge maintained by the dialogue manager. It is also possible that the differing information was simply not significant enough to cause a change in the dialogue state.

A difference was designated *better* if the new system performed better than the old system on that particular utterance. A difference could be better if it captured more information spoken by the user, or if it more correctly grasped the intention of the user by placing specific information into more appropriate semantic relationships. In a similar manner, a difference was designated *worse* if the old system performed better than the new system on that particular utterance.

Since the evaluation systems were run on input from log files, which reflected the output of the old system, the user replies to any system queries were fixed. If the context resolution in the new system were such that the system posed a different query than the corresponding one in the log file, the user's invariant reply would potentially be illogical. This would often instigate a nonsensical exchange of utterances in the evaluation system. When such a situation occurred, it was no longer possible to reliably compare the KV strings. These differences were *excluded* from the evaluation since the performance of neither system could confidently be assessed on the corresponding utterances.

The CR server was developed on a total of 60 dialogues, containing a total of 867 utterances. The development process involved the modification of the CR algorithm, and slight changes to the functions therein, to be run on the dialogues until all KV differences were shown to be *neutral* or *better*. The development results are shown in Table 5.1.

The CR algorithm was then frozen and tested on a set of 30 dialogues, containing a total of 586 utterances. The test results, also shown in Table 5.1, were very promising. Ten (1.7 percent) of the total test utterances were excluded, leaving 576 utterances. Both the old and the new system performed identically on about 98.6 percent of these utterances, about 1.0 percent were handled better by the new system, and less than

| Set | *Utterances* | *Better* | *Neutral* | *Worse* | *Excluded* |
|---|---|---|---|---|---|
| Dev | 867 | 15 | 27 | 0 | 16 |
| Test | 586 | 6 | 23 | 2 | 10 |

Table 5.1: Results showing the number of key-value differences between the CR server and the old CR component on 60 dialogues in the development set and on 30 dialogues in the test set.

0.4 percent were handled better by the old system.

Most interesting is how the new system was able to perform better than the old system on several utterances; some examples will soon be given. First, however, it must be explained why, and under what circumstances, the old system performed better than the new system and if, in fact, this is really significant.

## 5.3 Worse Performance

A new feature of the parser, as explained in Section 4.5, is to retain "extraneous" topics from a user's utterance in an `:and_topic` chain in the semantic frame. This is merely a design decision, but one needs to consider whether this information should truly be retained, especially given the fact that much of the information in the semantic frame could be incorrectly hypothesized by the speech recognizer.

The following example spotlights this situation, in which a hypothesized topic forms a relationship and the truly spoken topic gets buried in the parse and is, ultimately, unresolved.

Consider this utterance, spoken by a user:

"Edinburgh to Amsterdam KLM UK United Flight twenty eighty two"

and the corresponding hypothesis from the speech recognizer:

"denver and amsterdam k l m airline as flight twenty eighty two"

The NL server then generates a semantic frame for the hypothesized utterance and sends it to the CR server. The semantic frame generated by the new parser is shown in Figure 5-2.

The hypothesized `:topic` city, *denver*, is part of a relationship, *flight ambiguous city*, and it is satisfied. This is the same relationship that the truly spoken `:and_topic`

```
{c what_about
   :topic {q flight
             :pred {p flight_number
                      :topic 2082 }
             :pred {p ambiguous
                      :topic {q city
                                :name "denver"
                                :and_topic {q city
                                              :name "amsterdam" } } }
             :pred {p airline
                      :topic {q airline_name
                                :name "k l m" } } } }
```

Figure 5-2: The new system's semantic frame for "denver and amsterdam k l m airline as flight twenty eighty two."

```
{c display
   :domain "Mercury"
   :topic {q flight
             :pred {p flight_number
                      :topic 2082 }
             :pred {p ambiguous
                      :topic {q city
                                :name "denver"
                                :and_topic {q city
                                              :name "amsterdam" } } }
             :pred {p airline
                      :topic {q airline_name
                                :name "k l m" } }
             :pred {p source
                      :topic {q city
                                :name "EDI" } }
             :pred {p destination
                      :topic {q city
                                :name "AMS" } } } }
```

Figure 5-3: The new system's semantic frame for "denver and amsterdam k l m airline as flight twenty eighty two" after context resolution. The `:and_topic`, *amsterdam*, failed to be resolved.

city, *amsterdam*, would like to form with *flight*. However, the knowledge representation is such that only a single *flight ambiguous X* relationship can exist. The CR server fails to resolve the `:and_topic`, as shown in the resolved semantic frame in Figure 5-3.

In the new parser, the first fragment topic is chosen as the user-intended topic and any subsequent such topics are deemed "extraneous." This forces us to wonder whether the reverse would be a more optimal parsing strategy, that is, whether the most recently mentioned fragment topic should be viewed as user-intended, and any previous such topics as "extraneous." Such a modification would improve performance in this example, but could potentially be detrimental for other utterances. Further research is required to determine which parsing strategy is more optimal.

The old system utilizes the old parser, which generates the parsed semantic frame as shown in Figure 5-4. The typical behavior of the old parser is to discard all but the most recently mentioned fragmentary topic, for which it establishes a relationship in the semantic frame [39]. This is based on the theory that the most recently mentioned fragmentary topic is possibly a correction and, therefore, should be viewed as the user-intended topic. Consequently, the parser did not retain the "extraneous" `:topic` city *denver*, and *amsterdam* correctly forms the *flight ambiguous city* relationship, even after context resolution. The resolved semantic frame is shown in Figure 5-5.

The KV strings for the old and the new system are shown in Figure 5-6. Notice that the only difference is the value of the `city` key which represents an ambiguous city. The system reply to this utterance is actually the same in both systems since the ambiguous city difference was not significant enough to change the dialogue strategy. This KV difference was not counted as *neutral*, however, since unresolved information in an `:and_topic` could become quite a significant problem. This issue must, therefore, be addressed in future versions of the CR server.

There was another KV difference designated *worse*. This difference, however, was not the result of differing functionality between the two systems. Rather, it was the result of an incomplete inheritance specification in the external file for the new CR server, which could be fixed simply by adding the necessary constraint.

```
{c what_about
   :topic {q flight
            :pred {p flight_number
                     :topic 2082 }
            :pred {p ambiguous
                     :topic {q city
                               :name "amsterdam" } }
            :pred {p airline
                     :topic {q airline_name
                               :name "k l m" } } } }
```

Figure 5-4: The old system's semantic frame for "denver and amsterdam k l m airline as flight twenty eighty two."

```
{c display
   :domain "Mercury"
   :topic {q flight
            :pred {p flight_number
                     :topic 2082 }
            :pred {p ambiguous
                     :topic {q city
                               :name "amsterdam" } }
            :pred {p airline
                     :topic {q airline_name
                               :name "k l m" } }
            :pred {p source
                     :topic {q city
                               :name "EDI" } }
            :pred {p destination
                     :topic {q city
                               :name "AMS" } } } }
```

Figure 5-5: The old system's semantic frame for "denver and amsterdam k l m airline as flight twenty eighty two" after context resolution.

New system KV string:
```
clause: display destination: AMS source: EDI city: DEN
flight_number: 2082 airline: KL
```

Old system KV string:
```
clause: display destination: AMS source: EDI city: AMS
flight_number: 2082 airline: KL
```

Figure 5-6: KV strings for “denver and amsterdam k l m airline as flight twenty eighty two” after context resolution in the new and old system.

## 5.4 Neutral Performance

The majority of KV differences in both the development and the test sets were designated *neutral*. Most of them were not very interesting since the differences were very slight and insignificant, such as a small variation in the confidence score for a specific concept, or the insertion or deletion of a single concept, such as `:quantifier` which is more for featural completeness than for determining the course of the dialogue. Since there are more interesting phenomena occurring in those utterances for which the new system performed better, attention will be focused on that area.

## 5.5 Better Performance

There are several utterances for which the new system performed better than the old system. We will briefly describe some general areas of improvement observed throughout the development and evaluation of the CR server. This will be followed by an elaboration on two specific and outstanding example utterances for which the new CR server outperformed the old CR component.

### 5.5.1 General Improvements

The sequence of function execution in the old CR component was such that, in some circumstances, information that should not have been propagated from the history was, in fact, inherited. The sequence in the CR server was modified to appropriately handle this issue. The new sequence is also specified as generally as possible, so that it may, potentially, correctly handle some unforeseen situations.

Another oversight in the old CR component was that information from the history could be multiply inherited into the current semantic frame. This caused problems in some instances. The new CR server handles this by marking each piece of information inherited from the history. In this way, the current semantic frame may only inherit an unmarked piece of information from the history record.

The processing of the old CR component was such that, in a few circumstances, the current semantic frame would be left containing multiple sources, destinations, or other necessarily singular concepts. The functionality of the new CR server, especially the functions that reorganize unsatisfied objects within the semantic frame, tended to produce "clean" frames, in that there was only one instance of each necessarily singular concept.

The new CR server supports the resolution of multiple mouse clicks, whereas the old CR component could only handle a single click. Our dialogue systems currently allow a user to make two click references, which especially facilitates asking for directions in the VOYAGER city guide domain. When presented with a map, a user may, for example, click on two map locations and say, "How do I get from here to there?" The new CR server is able to resolve such an utterance, whereas the old CR component could not. This feature was not evaluated, but it is a feature that we hope will be useful in future research.

We have just presented some general improvements of the new CR server over the old CR component. In order to demonstrate the improved power of the new server, however, we now present two specific examples of context resolution taken from the MERCURY evaluation dialogues.

### 5.5.2 Example 1

In the first example, the user spoke the following utterance:

"I would like to fly from Detroit, Michigan on November sixth to San Francisco, California."

The speech recognizer produced the following hypothesis:

"i would like to fly from detroit michigan on november sixth in san francisco california"
in which "*to* san francisco" was substituted with "*in* san francisco." Already, there is an error by the recognizer that will, in the best case, be fixed by the parser. However, both the old and the new parser produce the semantic frame shown in Figure 5-7, in which the nonsense relationship, *city in city*, has been established.

The old CR component is unable to resolve the *city in city* relationship, so the relationship is left alone. This resolved frame is shown in Figure 5-8. The only modifications to the parsed frame were the promotions of the predicate chains *depart-source* and *depart-when-month_date*. When the dialogue manager receives this resolved frame, it is unable to identify *san francisco* as a meaningful concept, due to the *city in city* nesting, so the system replies with "What city does the flight arrive in?" It would appear to the user that the system was oblivious to the fact that the destination had already been supplied.

Given the parsed semantic frame, in Figure 5-7, the new CR server identifies that the *city in city* relationship is not valid and that *san francisco* is an unsatisfied topic. The "Organize Topics" stage tries to find a relationship in which this *city* topic will be satisfied. The server consults the external file and establishes the *flight ambiguous city* relationship for the *san francisco* topic. The new resolved semantic frame is shown in Figure 5-9.

When the dialogue manager receives this semantic frame, it discovers the ambiguous city, *san francisco*. Since a destination city is still required, the dialogue manager realizes that this ambiguous city could be the intended destination city. In its discourse update, the dialogue manager sends a *destination* predicate to be incorporated into the history record. The dialogue manager also sends an *airline* predicate in the discourse update which must come from longer-term information maintained by the dialogue manager. Following the discourse update, the semantic frame in Figure 5-10 represents the reconstructed intention of the user, salvaged by collaboration between the CR server and the dialogue manager.

From this example, it can be seen how the new CR server performs better than the old CR component by going deep into the semantic frame and verifying that every

```
{c statement
:topic {q flight
      :pred {p depart
           :pred {p source
                :topic {q city
                      :name "detroit"
                      :pred {p in
                           :topic {q city
                                 :name "san francisco"
                                 :pred {p in
                                      :topic {q state
                                      :name "california" } } } } } }
           :pred {p when
                :pred {p month_date
                     :topic {q date
                           :day_number 6
                           :month "november" } } } } } }
```

Figure 5-7: The old and new systems' parsed semantic frame for "i would like to fly from detroit michigan on november sixth in san francisco california."

```
{c statement
:topic {q flight
       :pred {p source
             :topic {q city
                    :name "detroit"
                    :pred {p in
                          :topic {q city
                                 :name "san francisco"
                                 :pred {p in
                                       :topic {q state
                                              :name "california" } } } } } }
       :pred {p month_date
             :topic {q date
                    :day_number 6
                    :month "november" } } } }
```

Figure 5-8: The old system's resolved semantic frame for "i would like to fly from detroit michigan on november sixth in san francisco california."

```
{c statement
  :topic {q flight
          :pred {p source
                  :topic {q city
                           :name "detroit" } }
          :pred {p month_date
                  :topic {q date
                           :day_number 6
                           :month "november" } }
          :pred {p ambiguous
                  :topic {q city
                           :name "san francisco"
                           :pred {p in
                                   :topic {q state
                                            :name "california" } } } } } }
```

Figure 5-9: The new system's resolved semantic frame for "i would like to fly from detroit michigan on november sixth in san francisco california."

```
{c statement
  :topic {q flight
          :pred {p destination
                  :topic {q city
                           :name "san francisco"
                           :pred {p in
                                   :topic {q state
                                            :name "california" } } } }
          :pred {p source
                  :topic {q city
                           :name "DTW" } }
          :pred {p airline
                  :topic {q airline_name
                           :name "NW" } }
          :pred {p month_date
                  :topic {q date
                           :month "NOV"
                           :day_number 6
                           :day "tuesday" } } } }
```

Figure 5-10: The new system's semantic frame after discourse update for "i would like to fly from detroit michigan on november sixth in san francisco california."

topic and predicate is satisfied in its current location. If it is not satisfied, it will be moved to a more appropriate frame.

### 5.5.3 Example 2

In this example, the user spoke the following utterance:

"Like to get some pricing information from Dayton to Los Angeles."

The speech recognizer produced the following hypothesis:

"what is the price the information from cleveland to los angeles"

in which *Dayton* was substituted with *cleveland*, among other, more innocuous, errors. The old parser generates a semantic frame, as shown in Figure 5-11, in which *source* and *destination* are placed under the *fare* topic. The context-resolved frame is identical to the parsed frame. The dialogue manager can handle this semantic representation; however, it does not represent, semantically, what the user intended. The user asked for *information*, which does not even appear in the parsed frame. Semantically, *source* and *destination* are properties of a *flight*, not a *fare*. The new CR server attempts to more accurately portray these relationships in the semantic frame.

```
{c wh_question
   :topic {q fare
             :quantifier "which_def"
             :pred {p source
                      :topic {q city
                               :name "cleveland" } }
             :pred {p destination
                      :topic {q city
                               :name "los angeles" } } } }
```

Figure 5-11: The old system's parsed and resolved semantic frame for "what is the price the information from cleveland to los angeles."

The parsed semantic frame generated by the new parser is shown in Figure 5-12. Notice that *information* is retained as an `:and_topic`. The predicates, *source* and *destination*, have been placed under the *information* topic, but they may be reorganized in the appropriate stage of the CR algorithm. Several steps are taken to achieve the final resolved semantic frame. To demonstrate the transformation, the semantic frame after each relevant stage will now be shown.

The first relevant stage is "Organize Topics" in which the backward topic relationship, *information for fare*, is formed. The resulting frame is shown in Figure 5-13.

The next relevant stage is "Form Obligatory Topic Relationships" in which the obligatory topic relationship, *fare for flight*, is formed. The resulting frame is shown in Figure 5-14.

At this point, the predicates, *source* and *destination*, are unsatisfied under the *information* topic. Therefore, the next, and last, relevant stage is "Organize Predicates" in which *source* and *destination* are moved from the *information* topic to the *flight* topic where they are satisfied. The resulting semantic frame, shown in Figure 5-15, now represents what the user intended, "the information for the fare for the flight from (cleveland) to los angeles." While *cleveland* is not the city spoken by the user, it is correct as far as the CR server is concerned. In order to correct the city, the system must rely on the correction strategies of the dialogue manager. One possibility would be for the dialogue manager to use the recognition confidence scores of the cities to determine that *cleveland* is likely *not* the city spoken by the user. The dialogue manager may then confirm the city with the user.

## 5.6 Summary

The results of this evaluation are encouraging. The new CR server has achieved the goal of performing at least as well as the old CR component on the overwhelming majority of tested utterances. The CR server has also shown improved performance on a promising number of utterances in the flight reservation domain.

Another goal of the CR server was to provide a minimally equivalent mechanism for context resolution in all of the existing GALAXY domains. The groundwork has been laid for this, but extensive testing has only been performed in the MERCURY

```
{c wh_question
  :topic {q fare
           :quantifier "which_def"
           :and_topic {q information
                         :quantifier "def"
                         :pred {p source
                                 :topic {q city
                                          :name "cleveland" } }
                         :pred {p destination
                                 :topic {q city
                                          :name "los angeles" } } } }
```

Figure 5-12: The new system's parsed semantic frame for "what is the price the information from cleveland to los angeles."

```
{c wh_question
   :topic {q information
            :quantifier "def"
            :pred {p for
                     :topic {q fare
                              :quantifier "which_def" } }
            :pred {p source
                     :topic {q city
                              :name "cleveland" } }
            :pred {p destination
                     :topic {q city
                              :name "los angeles" } } } }
```

Figure 5-13: The new system's semantic frame for "what is the price the information from cleveland to los angeles" after forming the backward topic relationship, *information for fare*.

```
{c wh_question
   :topic {q information
            :quantifier "def"
            :pred {p for
                     :topic {q fare
                               :quantifier "which_def"
                               :pred {p for
                                        :topic {q flight} } } }
            :pred {p source
                     :topic {q city
                               :name "cleveland" } }
            :pred {p destination
                     :topic {q city
                               :name "los angeles" } } } }
```

Figure 5-14: The new system's semantic frame for "what is the price the information from cleveland to los angeles" after forming the obligatory topic relationship, *fare for flight*.

```
{c wh_question
:topic {q information
      :quantifier "def"
      :pred {p for
           :topic {q fare
                 :quantifier "which_def"
                 :pred {p for
                      :topic {q flight
                            :pred {p source
                                 :topic {q city
                                       :name "cleveland" } }
                            :pred {p destination
                                 :topic {q city
                                       :name "los angeles" } } } } } } } }
```

Figure 5-15: The new system's completely resolved semantic frame for "what is the price the information from cleveland to los angeles" after moving *source* and *destination* to the *flight* topic from the *information* topic.

domain. Nevertheless, given the increased capabilities that the new CR server puts at our disposal, powerful modifications to any domain's context resolution functionality can ideally be accomplished solely by adding detailed constraints to the external file.

# Chapter 6

# Summary and Future Work

## 6.1 Summary

Throughout this thesis, we have presented the new Context Resolution (CR) server for the GALAXY conversational system framework. The purpose of the CR server is twofold—to attempt to reconstruct a user's intention in the case of a recognition error or a robust parse, and to interpret an individual utterance in context.

This section will summarize the work presented in the previous chapters, beginning with the definition of context resolution and ending with the results obtained from our evaluation of the CR server.

### 6.1.1 Context Resolution

In Chapter 1, we defined *context resolution* to be the process by which one dialogue participant interprets another participant's utterance. Several sources, including the dialogue history, physical and temporal context, inference, shared world knowledge, and common sense, may be used to produce a successful interpretation.

In terms of our spoken dialogue systems, the CR server uses the semantic representation of the previous utterance, a discourse entity list of previously mentioned topics, world knowledge via databases, and developer-specified constraints to interpret a user's utterance in context.

We motivated the new CR server with the goals of domain-independence, extensibility, intention reconstruction, and support for a key-value-based knowledge representation. Several sample dialogues were presented, demonstrating the necessity for the functionality provided by the new CR server.

In Chapter 2, we described the related studies of *dialogue* and *discourse*, the history of discourse processing in conversational systems, as well as how context resolution can be considered a subset of discourse processing.

### 6.1.2 Structural Improvements

In Chapter 3, we described several improvements to the framework supporting context resolution in the new CR server. The first of these is the creation of the NETWORK and NODE structures, along with a new meta-language, to allow constraints to be specified on the basis of the complete context of a semantic frame. This lets the developer specify detailed constraints, which is not possible in the old CR component. This improvement also supports our notion of *semantic satisfaction*, in which each object in a frame is either *satisfied* or *unsatisfied*. This increased power, for example, allows one to specify that a given predicate is satisfied *only* if it has a specific parent.

Another improvement is the modified history record. This structure contains the semantic representation of the previous user utterance, as well as a bounded list of all topics introduced throughout the dialogue. This list is subsequently used for reference resolution. The old CR component stores the semantic representation of the previous utterance, along with selected concepts in semantic slots; however, no extended history is maintained. The new CR server allows the user to make references to entities introduced into the dialogue from several utterances back.

The third major improvement is external function sequence control of the CR algorithm by a *dialogue control table*. This previously existing mechanism in the GALAXY framework allows the sequence of function execution within the context resolution process to be modified externally, without recompilation of the code. This facilitates the development of the CR server, due to the very experimental nature of obtaining the proper order within the CR algorithm.

In Chapter 4, we began to describe each function in the context resolution process in detail. Each function contributes to at least one of the overall goals of the CR server—intention reconstruction or interpretation in context.

### 6.1.3 Intention Reconstruction

As mentioned above, one goal of the CR server is to attempt to reconstruct a user's intention in the case of a recognition error or a robust parse. This attempt is based on the notion of semantic satisfaction, in which all objects within a semantic frame representation are either satisfied or unsatisfied, according to developer-specified constraints. In the event of a recognition error, which the parser fails to resolve, the semantic frame representation of a user's utterance may contain predicates in locations where they are not satisfied. The CR server acknowledges the dissatisfaction of these predicates and attempts to move each one to a location where it will be satisfied.

The formation of topic relationships is another strategy used by the CR server to reconstruct a user's intention. A robust parse often results in several topic fragments. The old parser would retain only one of a set of such topics from a robust parse, discarding the others. This strategy is potentially problematic, since a meaningful concept may easily be lost. We made a slight modification to the parser, resulting in the retention of all topic fragments in a robust parse. The CR server, based on the notion that these individual topics may somehow be linked in a meaningful way, attempts to form relationships between them by consulting valid topic relationships specified in the external file.

If either of these strategies is successful, the CR server will have accomplished its goal of reconstructing the intention of the user. The old CR component did not possess any ability to handle such recovery from a potential recognition error or robust parse. It is our hope that this new strategy will create a more robust context resolution component, and a more natural dialogue for the user.

### 6.1.4 Interpretation in Context

The context in which an individual utterance is interpreted is comprised of several elements. These may include the successful reconstruction of a user's intention, the semantic representation of the previous utterance, a discourse entity list of topics introduced through the course of the dialogue, as well as world knowledge via databases.

Many of the functions comprising the new CR server are based on the functionality provided by the old CR component. While its overall functionality is similar, however, each function has been modified to be more powerful and completely domain-independent.

The reference resolution component still functions to determine the best antecedent for a given reference, but the new CR server allows the developer to specify, externally, what antecedents are valid for each possible reference. In the old CR component, features such as *gender* and *number* were "hard-coded" and inextensible. The new CR server also maintains an extended list of possible antecedents, allowing the user to reference entities from several utterances back.

The inheritance and masking utilities have retained their overall functions to propagate and to forget information from the history, respectively; however, the new CR server allows much more precise inheritance and masking constraints, based on the context of descendant and ancestor frames, to be specified.

Another related feature of the new CR server is the pragmatic verification of relationships that result from inheritance. This utility facilitates the incorporation of world knowledge via databases into the context resolution process.

The ellipsis and fragment resolution component still interprets elliptical phrases (e.g., "What about Boston?") and utterance fragments (e.g., "Tomorrow") in context. However, the new NETWORK and NODE data structures facilitate access to the historical context to find an optimal location in which to insert the phrase or fragment, resulting in a more appropriate resolution.

All of these improvements contribute to a more powerful and more accurate context resolution component, which we hope will be useful for future research.

### 6.1.5 SpeechBuilder

In Chapter 4, we described the new CR server's ability to handle context resolution, based on a knowledge representation of key-value pairs, for the SPEECHBUILDER software tool. Currently, in domains created with SPEECHBUILDER, the new CR server supports the inheritance and masking of historical information, as well as *discourse updates* and *system initiatives* from the dialogue manager. We hope to extend context resolution support for these domains; the challenge lies in providing powerful functionality that remains simple and intuitive to control for a non-expert developer.

### 6.1.6 Evaluation

In Chapter 5, we presented an evaluation of the new CR server, which consisted of reprocessing offline a sequence of utterances contained in log files from user interactions with MERCURY. Each log file was run through the system twice—once using the old CR component, and once using the new CR server. The corresponding key-value strings from each run were compared to measure performance. About 1.7 percent of the total test utterances were excluded due to the consequent incoherence of the associated dialogues. The new CR server performed identically to the old CR component on about 98.6 percent of the remaining utterances, while about 1.0 percent were handled better by the new CR server, and less than 0.4 percent were handled better by the old CR component.

We consider these results to be very promising, especially since the increased power of the new CR server was not thoroughly utilized in the evaluation. We have yet to take full advantage of the new framework and constraint specification offered by the new CR server in our current domains. We believe that the addition of more detailed constraints, impossible in the old CR component, will serve to improve the performance of context resolution as a whole.

## 6.2 Future Work

There is a lot of potential work for improving the CR server. This section will identify and briefly describe only some of the many possibilities.

### 6.2.1 Temporal Deixis

The dialogue manager still handles the resolution of temporal deixis, such as "tomorrow" and "the following Friday," to absolute dates. This functionality would more appropriately be handled during context resolution. Current work in SLS is developing a generic and independent date/time server [38], which could potentially be called from the CR server to resolve temporal deixis, thus, relieving the dialogue manager of this resolution task.

### 6.2.2 Complex Constraints

The new framework and meta-language for the CR server has provided the developer with a very powerful means by which to specify complex context resolution constraints. The range of this power has not been extensively studied; however, the development of new domains will allow us to fully explore and to take full advantage of the new constraint specification from the initial development of each domain's context resolution component.

### 6.2.3 Reference Resolution Algorithm

The current reference resolution algorithm in the CR server simply searches for the first discourse entity in the DE list that satisfies given constraints. While this algorithm is currently sufficient, we would like to revise the algorithm to better handle competing antecedents. Perhaps the DE list could be divided into focus spaces, so that the server would be aware of all entities in focus during a given interval. This may help facilitate ambiguity resolution.

### 6.2.4 Automatic Constraint Generation

SPEECHBUILDER automatically creates a set of default context resolution constraints. The developer is subsequently able to modify these constraints to add limited complexity to the domain. In our traditional domains (i.e., JUPITER, MERCURY, ORION, PEGASUS, and VOYAGER), many more types of constraints featuring greater complexity must be specified by the developer. Automatically generating these constraints would be extremely difficult. However, perhaps there is some way in which common relationships between semantic objects may be learned to produce a default set of, for example, **topic-predicate** satisfication constraints or topic relationships. Such a utility would be much more efficient, relieving the developer of the cumbersome task of specifying the multitude of compulsory constraints.

### 6.2.5 Beyond Context Resolution

We believe the generic nature of the NETWORK and NODE structures, and the constraint specification meta-language, will allow this new mechanism to be extended to any system component that may need to verify the contents within a semantic frame. For instance, this utility might prove to be useful for translation tasks. Among natural languages, the linguistic structure for a single meaning may differ. Consider the query, "What is your name?" In English, *name* may be designated the highest-level object in the semantic frame representation. In Mandarin, the translation is "ni3 jiao4 shen2 me5 ming2 zi4?" ("You call what name?"). Here, the *call* predicate may be designated the highest-level object. The semantic frame representation of this utterance will be different for each language, due to the differing linguistic structures. The utilities of the new CR server are able to verify the structure of a given semantic frame and, using satisfaction constraints, might be able to transform the semantic frame representation for one language into the semantic frame representation for the other. The CR server contains the potential for this functionality, and we hope to be witness to a more in-depth exploration of this issue.

## 6.3 Final Words

We set out to create a new and independent Context Resolution server for the GALAXY framework, with the goal that it would perform at least as well as the old context resolution component. We believe that this minimal goal has been fulfilled and, ahead of us, lies the opportunity to expand and improve the server even further.

Discourse processing and dialogue strategy are closely intertwined. We previously defined each notion separately while, in reality, their effect on one another is significant. Exactly how these duties should be divided within a spoken dialogue system is highly varied within current implementations and there is not much agreement on the optimal approach. In our systems, the CR server handles context resolution, which is a subset of discourse processing, while the dialogue manager handles dialogue strategy, as well as the remaining discourse processing tasks. Whether or not this is the best way to divide the understanding component of our systems is uncertain. Perhaps, for example, an entirely separate component is needed to handle the remaining discourse processing duties, possibly involving user goals and intentions.

As previously mentioned, SLS is currently working on a generic dialogue manager, which can handle simple dialogue strategies in any domain. We plan to continue such research and to more profoundly explore the interrelations between context resolution, discourse processing, and dialogue strategy. During this study, we may discover that our context resolution process is incomplete, requiring more functionality to deal with issues such as intention recognition or presupposition failure—in other words, context resolution may cross over into the realm of "meaning across several utterances," rather than strictly handling the interpretation of an individual utterance in context.

On the other side of the coin, we may discover that the new intention reconstruction utility of the CR server is successful enough to be its own independent component. In either of the above cases, however, the notion of context resolution would require redefinition.

Nevertheless, the current division of our dialogue, discourse, and context resolution functionality has contributed to significant success in the three processes' collaborative role of natural language understanding.

Overall, we are very encouraged by the evaluation results that the new CR server performed as well as the old CR component on the majority of utterances. We look forward to further experimentation with the power of the new constraint meta-language, and we hope that the new CR server, in conjunction with the dialogue manager, will provide an improved and more natural experience for each user of our spoken dialogue systems.

# Appendix A

# Constraint Specification Meta-Language

This appendix contains information on using the constraint specification meta-language. The purpose of this meta-language is to allow a developer to easily specify conditions to be tested in a given semantic frame. For example, we can send the following frame:

```
{c top_frame
   :clause {c statement
              :topic {q season
                        :name "summer"
                        :pred {p come
                                 :mode "ing" } } } }
```

along with a constraint, to the testing utility which verifies whether or not the condition is true for this frame. Consider these constraints:

```
:clause = statement -> :topic = season          True
:clause -> (:topic -> :name = spring | summer)  True
:clause -> (:topic -> :name = autumn | winter)  False
```

The following sections will describe the syntax and semantics for this language. Note that, in the following descriptions, `:key`, `string`, `substring`, `integer`, and `float` are all-inclusive terms for any keyword, string, substring, integer, or float, respectively.

## A.1 Reserved Words

- **:topic**

  This specifies a **topic** object. It may be a string, integer, float, or frame. If it is a frame object, the type of the frame *must* be **topic**.

- **:pred**

  This specifies a **predicate** object, which is always a frame. Any single frame may have multiple `:pred` objects.

- **:PARENT**

  This specifies a parent frame object. Its frame type may be any of **clause**, **topic**, or **predicate**.

- **:P_PARENT**

  This specifies a parent frame object with the **predicate** frame type.

- **:T_PARENT**

  This specifies a parent frame object with the **topic** frame type.

- **:C_PARENT**

  This specifies a parent frame object with the **clause** frame type.

- **:CLAUSE**

  This specifies the nearest encompassing frame of type **clause**.

## A.2 Existence

- **:topic**

  The given frame contains the `:topic` key.

- **:pred**

  The given frame contains one or more `:pred` frames.

- **:PARENT**

  The given frame has a parent frame.

- **:P_PARENT**

  The given frame has a parent frame of type **predicate**.

- **:T_PARENT**

  The given frame has a parent frame of type **topic**.

- **:C_PARENT**

  The given frame has a parent frame of type **clause**.

- **:CLAUSE**

  The given frame has an ancestor frame of type **clause**.

- **:key**

  The given frame contains the specified keyword.

### A.2.1 Negation

Any of the above tokens may be negated with the "**!**" prefix, to indicate the *non*-existence of each token in the given frame. For example:

- **!:pred**

  The given frame does not contain any `:pred` frames.

- **!T_PARENT**

  The given frame does not have a parent frame of type **topic**.

## A.3 Frame Name

Given that the specified `:key` exists in the given frame, and given that the object represented by `:key` is a frame object, conditions may be imposed on the frame object's name.

### A.3.1 Equality

- `:key = string`

  The name of the frame must be `string`.

- `:key = %substring`

  The name of the frame must *contain* `substring`.

### A.3.2 Inequality

- `:key < string`

  The name of the frame must occur alphabetically *before* `string`.

- `:key <= string`

  The name of the frame must occur alphabetically *before* `string`, or the name must be equal to `string`.

- `:key > string`

  The name of the frame must occur alphabetically *after* `string`.

- `:key >= string`

  The name of the frame must occur alphabetically *after* `string`, or the name must be equal to `string`.

### A.3.3 Negation

Any of the above operations may be negated with the **!** prefix, to result in: **!**=, **!**<, **!**<=, **!**>, **!**>=.

## A.4 Integers

Given that the specified `:key` exists in the given frame, and given that the object represented by `:key` is an integer object, conditions may be imposed on the integer's value.

### A.4.1 Equality

- `:key = integer`

  The value of the integer must equal `integer`.

### A.4.2 Inequality

- `:key < integer`

  The value of the integer must be less than `integer`.

- `:key <= integer`

  The value of the integer must be less than or equal to `integer`.

- `:key > integer`

  The value of the integer must be greater than `integer`.

- `:key >= integer`

  The value of the integer must be greater than or equal to `integer`.

### A.4.3 Negation

Any of the above operations may be negated with the `!` prefix, to result in: `!`=, `!`<, `!`<=, `!`>, `!`>=.

## A.5 Floats

Given that the specified `:key` exists in the given frame, and given that the object represented by `:key` is a float object, conditions may be imposed on the float's value.

### A.5.1 Equality

- `:key = float`

  The value of the float must equal `float`.

### A.5.2 Inequality

- `:key < float`

  The value of the float must be less than `float`.

- `:key <= float`

  The value of the float must be less than or equal to `float`.

- `:key > float`

  The value of the float must be greater than `float`.

- `:key >= float`

  The value of the float must be greater than or equal to `float`.

### A.5.3 Negation

Any of the above operations may be negated with the `!` prefix, to result in: !=, !<, !<=, !>, !>=.

## A.6 Conjunctions

The conjunctive operator (`&`) may be used to specify a conjunction between two conditions. For example:

- `:key < value & :pred`

  The value of `:key` is less than `value` *and* one or more `:pred` frames exist in the given frame.

- `:key != value & !:topic`

  The value of `:key` is not equal to `value` *and* the `:topic` key does not exist in the given frame.

## A.7 Disjunctions

The disjunctive operator (`|`) may be used to specify a disjunction between two conditions, as well as between two values. For example:

- `:key < value | :pred = pvalue`

  The value of `:key` is less than `value` *or* a `:pred` frame named `pvalue` exists in the given frame.

- `:key = value1 | value2 | value3`

  The value of `:key` equals `value1` *or* `value2` *or* `value3`.

## A.8 Hierarchical Constraints

### A.8.1 Descendants

The "`->`" symbol may be used following the key representing a frame object to impose conditions inside that frame. For example:

- `:key -> :descendant = value`

  The given frame contains the `:key` keyword, which represents a frame object *and* this frame contains the `:descendant` keyword, which has a value equal to `value`.

- `:key = value -> !:descendant`

  The given frame contains the `:key` keyword, which represents a frame object that is named `value` *and* this frame does not contain the `:descendant` keyword.

### A.8.2 Ancestors

The "`->`" symbol may be used following a key representing *any* object to impose conditions on that object's parent, grandparent, etc. For example:

| Operator | Operator Name |
|---|---|
| `(...)` | Scope |
| `%` | Substring |
| `!` | Negation |
| `->` | Hierarchy |
| `\|` | Disjunction |
| `&` | Conjunction |
| `=, <, <=, >, >=` `!=, !<, !<=, !>, !>=` | Equality and Relations |

Table A.1: Operator precedence, shown in descending order, for the constraint specification meta-language.

- `:key -> :PARENT = value`

  The given frame contains the `:key` keyword, and `:key`'s parent frame is named `value`.

- `:key -> :CLAUSE = value`

  The given frame contains the `:key` keyword, and the nearest ancestor clause frame of this key is named `value`.

## A.9 Operator Precedence

The precedence of the operators is shown in descending order in Table A.1. The operator precedence must be respected when constraints are specified. To modify operator precedence, parentheses must be placed around the relevant portion of the constraint. Some examples follow, based on the frame in Figure A-1.

If the current scope is the clause frame named `truth`:

- `:topic = flight -> :pred = airline & :pred = arrival_time`

  is *false* since the default association is

  `(:topic = flight -> :pred = airline) & :pred = arrival_time`

  and `:pred = arrival_time` does not exist in the frame named `truth`.

- `:topic = flight -> (:pred = airline & :pred = arrival_time)`

```
{c truth
  :topic {q flight
          :pred {p airline
                  :topic "american" }
          :pred {p arrival_time
                  :topic {q time
                           :quantifier "which"
                           :pred {p source
                                   :topic {q city
                                            :name "denver" } } } } } } }
```

Figure A-1: Semantic frame for "When is the American flight from Denver arriving?"

is *true* since the parentheses associate both `:pred`s under the frame named `flight`, where they both exist.

If the current scope is the clause frame named `time`:

- `:quantifier = %which & :pred -> :P_PARENT | :T_PARENT`
  is *false* since the default association is
  `:quantifier = %which & ((:pred -> :P_PARENT) | :T_PARENT)`
  and the `time` frame does not have a `:pred` that has a predicate parent frame nor does the `time` frame have a topic parent frame.

- `:quantifier = %which & (:pred -> (:P_PARENT | :T_PARENT))`
  is *true* since the parentheses associate the constraint such that `:pred` may have either a predicate parent frame or a topic parent frame. Since either `:pred` has the `time` topic frame as its parent, and the rest of the constraint is satisfied, the entire constraint is *true*.

# Bibliography

[1] J. Allen. *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1995.

[2] J. Allen, G. Ferguson, and A. Stent. An architecture for more realistic conversational systems. In *Proc. Intl. Conf. on Intelligent User Interfaces*, pages 1–8, Santa Fe, New Mexico, January 2001.

[3] L. Baptist and S. Seneff. GENESIS-II: A versatile system for language generation in conversational system applications. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 271–274, Beijing, China, October 2000.

[4] D. Bobrow, R. Kaplan, M. Kay, D. Norman, H. Thompson, and T. Winograd. GUS: A frame-driven dialog system. *Artificial Intelligence*, 8:155–173, 1977.

[5] R.A. Bolt. Put-That-There: Voice and gesture at the graphics interface. *Computer Graphics*, 14(3):262–270, 1980.

[6] E. Bos, C. Huls, and W. Claassen. EDWARD: Full integration of language and action in a multimodal user interface. *International Journal of Human-Computer Studies*, 40:473–495, 1994.

[7] D.K. Byron. Improving discourse management in TRIPS-98. In *Proc. European Conf. on Speech Communication and Technology*, pages 1379–1382, Budapest, Hungary, September 1999.

[8] M.S. Carberry. A pragmatics based approach to understanding intersentential ellipsis. In *Proc. Mtg. of the Assoc. for Computational Linguistics*, pages 188–197, Chicago, Illinois, 1985.

[9] J. Cassell, T. Bickmore, M. Billinghurst, L. Campbell, K. Chang, H. Vilhjálmsson, and H. Yan. Embodiment in conversational interfaces: Rea. In *Proc. Conf. of the Assoc. for Computing Machinery (ACM) Special Interest Group on Computer-Human Interaction (SIGCHI)*, pages 520–527, Pittsburgh, Pennsylvania, May 1999.

[10] J. Cassell, T. Bickmore, L. Campbell, H. Vilhjálmsson, and H. Yan. More than just a pretty face: Conversational protocols and the affordances of embodiment. *Knowledge-Based Systems*, 14:55–64, 2001.

[11] N. Chovil. Discourse-oriented facial displays in conversation. *Research on Language and Social Interaction*, 25:163–194, 1991.

[12] R.E. Cullingford. SAM. In R.C. Schank and C.K. Riesbeck, editors, *Inside Computer Understanding: Five Programs Plus Miniatures*, pages 75–119. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1981.

[13] J. Dowding, E.O. Bratt, and S. Goldwater. Interpreting language in context in CommandTalk. In *Communicative Agents: The Use of Natural Language in Embodied Systems*, pages 63–67, Seattle, Washington, May 1999. ACM Special Interest Group on Artificial Intelligence (SIGART).

[14] M. Eckert and M. Strube. Resolving discourse deictic anaphora in dialogues. In *Proc. of the European Chapter of the Assoc. for Computational Linguistics*, pages 37–44, Bergen, Norway, June 1999.

[15] G. Ferguson and J. Allen. TRIPS: An integrated intelligent problem-solving assistant. In *Proc. of the National Conference on Artificial Intelligence*, pages 567–572, Madison, Wisconsin, July 1998. American Association for Artificial Intelligence.

[16] J. Glass, J. Chang, and M. McCandless. A probabilistic framework for feature-based speech recognition. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 2277–2280, Philadelphia, Pennsylvania, 1996.

[17] J. Glass, G. Flammia, D. Goodine, M. Phillips, J. Polifroni, S. Sakai, S. Seneff, and V. Zue. Multilingual spoken-language understanding in the MIT VOYAGER system. *Speech Communication*, 17(1-2):1–18, March 1995.

[18] J. Glass, J. Polifroni, S. Seneff, and V. Zue. Data collection and performance evaluation of spoken dialogue systems: The MIT experience. In *Proc. Intl. Conf. on Spoken Language Processing*, Beijing, China, October 2000.

[19] J. Glass and E. Weinstein. SPEECHBUILDER: Facilitating spoken dialogue system development. In *Proc. European Conf. on Speech Communication and Technology*, pages 1335–1338, Aalborg, Denmark, September 2001.

[20] D. Goddeau, E. Brill, J. Glass, C. Pao, M. Phillips, J. Polifroni, S. Seneff, and V. Zue. GALAXY: A human-language interface to on-line travel information. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 707–710, Yokohama, Japan, September 1994.

[21] B.J. Grosz, A.K. Joshi, and S. Weinstein. Centering: A framework for modeling the local coherence of discourse. *Computational Linguistics*, 21(2):203–225, June 1995.

[22] B.J. Grosz, M.E. Pollack, and C.L. Sidner. Discourse. In M.I. Posner, editor, *Foundations of Cognitive Science*, chapter 11, pages 437–468. The MIT Press, Cambridge, Massachusetts, 1989.

[23] B.J. Grosz, D. Scott, H. Kamp, P. Cohen, and E. Giachin. Discourse and dialogue. In R. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue, editors, *Survey of the State of the Art in Human Language Technology*, chapter 6. Cambridge University Press, 1996. <http://cslu.cse.ogi/edu/HLTsurvey/>. Accessed 2 May 2002.

[24] B.J. Grosz and C.L. Sidner. Attention, intention, and the structure of discourse. *Computational Linguistics*, 12(3):175–204, 1986.

[25] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems*, 3(2):105–147, June 1978.

[26] J.R. Hobbs. On the coherence and structure of discourse. Report No. CSLI-85-37, CSLI, Stanford, California, October 1985.

[27] H. Kamp and U. Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory.* Kluwer Academic Publishers, Dordrecht, Germany, 1993.

[28] A. Kehler. A discourse copying algorithm for ellipsis and anaphora resolution. In *Proc. Conf. European Chapter of the Assoc. for Computational Linguistics*, pages 203–212, Utrecht, Germany, April 1993.

[29] L.B. Larsen, T. Brøndsted, H. Dybkjær, L. Dybkjær, B. Music, and C. Povlsen. State-of-the-Art of Spoken Language Systems – A Survey. Report 1 from the Danish Project in Spoken Language Dialogue Systems. STC Aalborg University, CCS Roskilde University, CST University of Copenhagen, September 1992.

[30] R. Lau, G. Flammia, C. Pao, and V. Zue. WebGALAXY: Beyond point and click– a conversational interface to a browser. *Computer Networks and ISDN Systems*, 29:1385–1393, 1997.

[31] O. Lemon, A. Bracy, A. Gruenstein, and S. Peters. The WITAS multi-modal dialogue system I. In *Proc. European Conf. on Speech Communication and Technology*, pages 1559–1562, Aalborg, Denmark, September 2001.

[32] D. Loehr. Hypertext and deixis. Presented and published by SIGMEDIA workshop entitled *Referring Phenomena in a Multimedia Context and Their Computational Treatment.* Association for Computational Linguistics. Madrid, Spain, July 1997.

[33] W.D. Mann and S.A. Thompson. Rhetorical Structure Theory: Towards a functional theory of text organization. *Text*, 8(3):243–281, 1988.

[34] K. McKeown. *Text Generation: Using Discourse Strategies and Focus Constraints to Generate Natural Language Text.* Cambridge University Press, Cambridge, United Kingdom, 1985.

[35] R. Moore, J. Dowding, H. Bratt, J.M. Gawron, Y. Gorfu, and A. Cheyer. CommandTalk: A spoken-language interface for battlefield simulations. In *Proc. Conf. on Applied Natural Language Processing*, pages 1–7, Washington, D.C., April 1997. Assocation for Computational Linguistics.

[36] K. Nagao and A. Takeuchi. Speech dialogue with facial displays: Multimodal human-computer conversation. In *Proc. Mtg. of the Assoc. for Computational Linguistics*, pages 102–109, Las Cruces, New Mexico, June 1994.

[37] Natural Born Kissers. Matt Selman, writer. Klay Hall, director. *The Simpsons.* Copyright FOX. Originally Aired 17 May 1998. Information obtained from <http://www.snpp.com/episodes/5F18> and <http://www.thesimpsons.com/episode_guide/>. Accessed 8 May 2002.

[38] J. Polifroni and G. Chung. *Promoting Portability in Dialogue Management.* Submitted to *Intl. Conf. on Spoken Language Processing*, Denver, Colorado, September 2002.

[39] S. Seneff. Robust parsing for spoken language systems. In *Proc. Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 189–192, San Francisco, California, March 1992.

[40] S. Seneff. TINA: A natural language system for spoken language applications. *Computational Linguistics*, 18(1):61–86, 1992.

[41] S. Seneff, C. Chuu, and D.S. Cyphers. ORION: From on-line interaction to off-line delegation. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 142–145, Beijing, China, October 2000.

[42] S. Seneff, D. Goddeau, C. Pao, and J. Polifroni. Multimodal discourse modelling in a multi-user multi-domain environment. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 192–195, Philadelphia, Pennsylvania, October 1996.

[43] S. Seneff, E. Hurley, R. Lau, C. Pao, P. Schmid, and V. Zue. GALAXY-II: A reference architecture for conversational system development. In *Proc. Intl. Conf. on Spoken Language Processing*, Sydney, Australia, November 1998.

[44] S. Seneff and J. Polifroni. Dialogue management in the MERCURY flight reservation system. In *Proc. ANLP-NAACL, Satellite Dialogue Workshop*, pages 1–6, Seattle, Washington, May 2000.

[45] A. Stent, J. Dowding, J. Gawron, E. Bratt, and R. Moore. The CommandTalk spoken dialogue system. In *Proc. Mtg. of the Assoc. for Computational Linguistics*, pages 183–190, College Park, Maryland, June 1999.

[46] K. Wauchope. Eucalyptus: Integrating natural language input with a graphical user interface. NRL Technical Report, NRL/FR/5510--94-9711, Navy Center for Applied Research in Artificial Intelligence, Naval Research Laboratory, Washington, D.C., 1994.

[47] B.L. Webber. So what can we talk about now? In M. Brady and R. Berwick, editors, *Computational Models of Discourse*, chapter 6, pages 331–371. MIT Press, Cambridge, Massachusetts, 1983.

[48] B.L. Webber. Structure and ostension in the interpretation of discourse deixis. *Natural Language and Cognitive Processes*, 6(2):107–135, 1991.

[49] T. Winograd. *Understanding Natural Language.* Academic Press, New York, New York, 1972.

[50] W.A. Woods. Semantics and quantification in natural language question answering. In B.J. Grosz, K.S. Jones, and B.L. Webber, editors, *Readings in Natural Language Processing*, pages 205–248. Morgan Kaufmann Publishers, Inc., Los Altos, California, 1986.

[51] WordNet ® 1.7, © 2001 Princeton University. G. Miller, Principal Investigator. <http://www.cogsci.princeton.edu/cgi-bin/webwn1.7.1?stage=1&word=deixis>. From Web WordNet ® 1.7.1. Accessed 5 May 2002.

[52] J. Yi, J. Glass, and L. Hetherington. A flexible, scalable finite-state transducer architecture for corpus-based concatenative speech synthesis. In *Proc. Intl. Conf. on Spoken Language Processing*, pages 322–325, Beijing, China, October 2000.

[53] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pao, T. Hazen, and L. Hetherington. JUPITER: A telephone-based conversational interface for weather information. *IEEE Transactions on Speech and Audio Processing*, 8(1):85–96, January 2000.

[54] V. Zue, S. Seneff, J. Polifroni, M. Phillips, C. Pao, D. Goodine, D. Goddeau, and J. Glass. PEGASUS: A spoken dialogue interface for on-line air travel planning. *Speech Communication*, 15(3-4):331–340, 1994.