A Computational Model of Turn-taking in Discourse

by

Toby Donaldson

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 1998

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis presents a computational model for turn-taking applicable to dialog-based interactive systems. A general, three-part model of the internal architecture of a turn-taking agent is presented, accounting for *why* an agent should take a turn, *what* it should contribute, and *when* it should act. For deciding what to say, the *next-action* constraint optimization problem is developed, which allows an agent to order multiple goals using an interruptible local search algorithm sensitive to the time-constraints imposed by dynamic dialog environments. For deciding when to speak, a formal account of turn-taking is given, based on the situation calculus, and accompanied by a state-based algorithm that tells an agent when it should take a turn based in part on the status of its processing. Two prototype implementations of course-advising systems are presented, to illustrate how interactive systems can be analyzed using a checklist for system designers derived from the turn-taking model. In developing a computational model for turn-taking, this thesis shows how to bring models for written discourse one step closer to spoken dialog, and thus advances the state of the art in natural language discourse theory and in natural language generation.

Acknowledgements

Many thanks to Robin Cohen, a good supervisor, friend, and teacher from whom I learned more than would fit into this thesis. Robin, thanks for all your help and support.

Many people provided helpful comments to me during my research. In particular, Cameron Shelley provided many insights along with introducing me to a number of other interesting areas of research in cognitive science. Marzena Makuta, who finished her thesis about a year before me, provided me with more ideas and food for thought than she might have realized at the time. Steve Woods, both at Waterloo and SFU, gave many helpful suggestions and encouragement for my particular line of research.

For the second half of 1996, I spent a fruitful six months working in the case-based reasoning group at Simon Fraser University. Thanks to Qiang Yang, Eddie Kim, Bonnie Cheng, and the rest of the CBR group and everyone else at SFU who made it such an enjoyable experience.

People who provided useful comments on my research include: Julian Fogel, Candy Sidner, Marilyn Walker, Peter Vanderheyden, Fahiem Bacchus, Forbes Burkowski, and Sandra Carberry.

I would like to thank the support staff in the UW computer science department, in particular Jane Prime, Wendy Rush, Debbie Mustin, Tracy Taves, Brenda McBay, and Ursula Thoene.

I would also like to acknowledge the tolerance of my eight former Waterloo officemates: Darcy Kroeker, Sovannary Tan, Elizabeth Cheryian, Sothi, Chris, Philip Fong, Kim Parsons, and Karsten Verbeurgt.

To everyone else who helped me along the way (and to those I've unintentionally forgotten to name), thanks for all your help. It was fun.

To Mom: thanks for all those tuna sandwiches.

Contents

List of Tables

List of Figures

Chapter 1

Turn-taking and Conversation

In the beginning, there were two tiny amoeba ... *Bugs Bunny*

Turn! Turn! Turn! *Pete Seeger*

1.1 Introduction: The Need for a Theory of Turn-taking

Turn-taking is one of the basic features of dialog: at most times, exactly one conversant has the "floor". To avoid both doubletalk (simultaneous speaking) and awkward pauses (no one speaking), conversants must share the floor and pass it between themselves smoothly. Turn-taking is in fact a fundamental skill that humans begin practicing at an early age:

> Turn-taking seems to be a phenomenon deeply rooted in human communication founded on a mutual awareness of sharing something. It has been found that, even at a very early stage in life, the child begins to acquire knowledge about this organization of togetherness and its characteristic patterning which exists in and builds up the dialogue. (Oreström 1983)

Oreström goes on to report on research that claims even in the very first week of life, young children are practicing interaction with eye glances and facial expressions, and, later, through "cooing" and "babbling" in response to adult talk. Halliday (1985) reports on films of mother-child interaction that show that it is often the *child* who initiates and controls these proto-conversations, which suggests that understanding of conversational initiative might also be ingrained from an early age. Since turn-taking is a natural and easy behaviour for people, we would like interactive computer systems to be equally skilled. A good turn-taker knows when to speak and when to listen. A good speaker knows when to offer someone else a chance to speak, and a good listener knows when to wait patiently, ask for a chance to speak, or to interrupt the current speaker. Not understanding how to share the conversational floor will lead to systems that are constantly stepping on the user's toes, providing more frustration than help.

1.1.1 An Example of Poor Turn-taking

Computer program interfaces are sometimes irritating because they do not take turns smoothly. The software development environment for an old version of Microsoft QBASIC[1] has a feature that allows the system to immediately warn the user about syntax errors. If the user moves the cursor from a syntactically incorrect line of code, then an error box pops up to describe the problem, and they must click a button with the mouse to make the box go away. This feature is useful in some situations, but many programmers turn it off and never use it again, because the interruptions are too frequent, unhelpful, and disruptive. For example, some programmers like to type block-like statements

[1] QBASIC is a software development package that includes a number of graphical-user interface features, such as automatic step-through debugging, windows with different views of the source code/output, extensive on-line help, etc. In later versions of QBASIC, the details of the interface have changed quite a bit so the particular problem described may no longer exist, but the underlying ideas and metaphors are still essentially the same.

in pairs, so they will not forget the end-block command. When writing a for-loop, the programmer might first type these two lines:

```
for

end for
```

Then after typing these two lines, the programmer fills in the for-loop condition and body. If the automatic syntax checking feature is enabled, this will cause a box to pop up that warns the programmer that the line containing `for` is incorrect, a fact that the programmer both knows and does not care about at this point. The problem with this particular program was that it stopped all the interaction and required that the user pay attention to this syntax error. In a way, this is like a child interrupting without an understanding of the whole situation. You can (sometimes) explain to the child that, while they are correct in realizing that this is an error, it is not helpful to point this out every time, because writing begin/end blocks is useful and the programmer has not actually made a mistake. While children learn, most computer programs do not, and must be replaced with new versions.

One way of describing the problem with the above interface is that it is taking a turn at the wrong time. A really smart system would watch the programmer, and realize that when she writes `for`/`end for` as above, she is carrying out a plan to construct a syntactically correct for-loop in a way that minimizes the chance of forgetting the `end for` line. Such a system could treat its interaction with the programmer like a kind of dialog or conversation, such that all the input from the user (e.g. typing, mouse clicks, etc.) counts as part of her "speech", and all the output from the system (e.g. writing messages to the screen, popping up windows, changing colours, etc.) are its utterances. Like a good human conversationalist, we want the system to know when it should speak, and what it should say. In the case of the programmer interface, it knows what to say (i.e. that there

is a syntax error), but since it always mentions this as soon as the cursor moves from the line, the system ends up irritating many users. A smart system should recognize the syntax error, but wait until the right time to tell the user. The right time might depend upon specific information about the user (e.g. are they an experienced programmer?), or on direct observation of the user's actions. For an experienced programmer, it will often only take a few seconds to fill in the details of the loop condition, and so after that might be a better time to report on a syntax error, if it still exists.

A gentler warning of a potential syntax error might also work. For example, instead of stopping all interaction and requiring the user to acknowledge the presence of the syntax error, the system could raise a less intrusive warning, one that the programmer could ignore if they so desire. Instead of popping up a dialog box, the interface could less obtrusively color the suspect line red to distinguish it from the other, correctly coded lines.

1.2 Overview

The main contribution of this thesis is to present a general, goal-oriented model of turn-taking. Throughout the thesis, we use the term *turn-taking agent* and assume that dialog conversants, whether people or computers, are agents. We will essentially follow the definition given by Russell and Norvig (1995), who define an agent as "...something that perceives and acts."(p.7). [2] In this thesis, the model of a turn-taking agent is divided into three parts, corresponding to the basic problems it must solve in any interaction:

[2]This is an extremely broad definition, and, for example, can be taken to include *any* computer system by treating the input as perception, and allowing internal state-changes to count as actions. Usually, one takes a pragmatic approach and only ascribes agenthood to things that act, or are intended to act, intelligently, in the sense that if a person had done the same thing, we would have normally said the action requires intelligence to perform. Agents are typically assumed to have internal mental states, such as beliefs, desires, and intentions, and their actions are determined by these states.

1. *Why* does it want to take a turn;

2. *What* does it want to contribute to the interaction;

3. *When* should it actually contribute to the interaction;

The first two parts correspond to problems that have long been recognized in computational discourse (and AI in general). As will be discussed in Chapter 2, most AI discourse systems recognize the need for the first two steps; for example, a natural language parser transforms strings of letters into a syntactic/semantic representation that the system can reason about. The second step captures the basic reasoning the system needs to do to make its decisions, and applying and understanding reasoning methods has perhaps been the major enterprise of AI since its inception: planning, theorem-proving, decision theory, heuristic search, etc. can all be seen as problem-solving techniques that an agent can use to achieve its goals (Russell and Norvig 1995).

The third step, however, represents a novel contribution to computational discourse. Spoken dialog is a time-pressured activity, and this makes a difference in the kinds of processing an agent can do and the output it produces. Computational discourse often assumes dialog to be *written*, like an essay or a letter. This is a helpful assumption, because spoken language is a different and often more complex medium; as Halliday (1985) points out, spoken language contains all the mistakes, cross-outs, re-starts, and disfluencies that a writer might make in a first draft, but then later edits out. While speakers can "edit" their speech, everything they do, mistakes included, will be there for everyone to see. The third step of the turn-taking architecture brings computational discourse a little closer to spoken language, but without letting in all the complexities of vocal speech. The conversational floor is a concept that comes from spoken language, and an awareness of it is ultimately an ability that will lead to better spoken dialog systems. We do not attempt to deal directly with spoken language in this thesis, since

the differences between spoken and written language are substantial enough that they are practically two different kinds of communication (Section 2.5). Our approach is to start from the perspective of written-language discourse, and then to take it a step closer to spoken language discourse by adding awareness of the floor.

By adding the concept of the floor to the written-discourse model, it becomes clear that the first two steps cannot be implemented in a way that ignores the new third step. In particular, the reasoning process used in step two requires some awareness of its processing state, in order for the agent to know when it is ready to speak/interrupt/etc. Our perspective on the three parts of a turn-taking agent can now be expanded as follows:

1. *Why* does it want to take a turn: agents are motivated by their environment, and their own deliberations. An agent might take a turn because someone has asked it a question, or because it has remembered a relevant piece of information, etc;

2. *What* does it want to contribute to the interaction: an agent will usually need to choose among multiple goals/actions when it is deciding what to say. It must not only try to make the best selection it can, it must also be sensitive to time constraints, especially in conversation where pauses can provide significant information;

3. *When* should it actually contribute to the interaction: an agent with something to say and a reason for saying it still needs to decide when it should act. A good turn-taking agent takes turns smoothly by making, and listening for, turn-yielding and turn-ending signals.

This thesis discusses how the three steps of turn-taking can be realized in a computational model. Many traditional AI formalisms do not worry about limited resources, and so in step two we use local search algorithms that work well under such resource constraints. For step three, we present a theory of turn-taking which imagines interacting

participants passing an abstract object called the *floor*[3] between them. This is formalized in the logic of turn-taking presented in Chapter 5.1, which provides an ontology and logical language for describing turn-taking situations.

In addition to providing a new approach to the modeling of AI dialog, this thesis presents some new directions for the design of interactive systems. For example, the idea of *initiative* is an important one in most multi-agent systems (Wooldridge and Jennings 1995). The core intuition shared by most researchers is the everyday usage of "initiative": one takes the initiative by starting something, by doing it volitionally or without prompting, and that there are different levels or degrees of initiative, as in the sense that some people show more initiative than others. In multi-agent systems, *mixed-initiative* intuitively means that more than one agent can have the initiative, in contrast to scripted systems where the roles of the agents are pre-set (Burstein and McDermott 1996).

Beyond this intuition, though, agreement on what initiative is, or what counts as a mixed-initiative system, diverges. Part of the confusion is due to a lack of a shared model of interaction among researchers, and a casual attitude towards fundamental issues such as turn-taking. Providing a clear, simple, and unambiguous model of turn-taking helps clarify the meaning and value of initiative and mixed-initiative systems.

1.3 Organization

The organization of this thesis is as follows. Chapter 2 covers some of the computational linguistics background. Chapter 3 introduces and outlines the general model for turn-taking agents, and discusses the initial motivation step for taking a turn. Chapter 4 and

[3]We use the term *floor* instead of *turn* to avoid some of the ambiguity that can arise when talking about the turn. For example, the phrase "He's taking a turn" is ambiguous; it could mean either that he is currently taking possession of the turn object from the speaker, or it could mean that he is currently the speaker.

Chapter 5 elaborate the second and third steps respectively. Chapter 4 treats action-selection as a kind of constraint optimization problem, and so introduces the general constraint satisfaction paradigm, and then presents local search algorithms for solving the action-selection problem. Chapter 5 introduces a formal account of turn-taking based on the situation calculus, and then gives a state-based algorithm for controlling a conversational agent's turn-taking behaviour. Chapter 6 gives an application of turn-taking to interactive system design. A general recipe for mapping an interactive system into a turn-taking form is given, which provides a designer with a way to systematically analyze an interactive system, which can lead to insights and generalizations. This is demonstrated with two sample course-advising systems: a command-line system called *SG*, and a graphically oriented system called SAM. Chapter 7 discusses related work, and finally Chapter 8 presents a summary of contributions and prospects for future research.

Chapter 2

Discourse Background

Man invented language to satisfy his deep need to complain. *Lily Tomlin*

2.1 Introduction

In this chapter, some of the linguistic background relevant to computational discourse will be covered. Brief mention will be made of natural language parsing and generation, followed by a more extensive discussion of discourse processing, focusing on Grosz and Sidner's model of discourse. We then discuss the speech-act approach to language processing, and summarize some of Oreström's work on turn-taking. Finally, we give a brief introduction to speech processing, in order to give the reader a feel for the issues that current speech processing systems face.

2.2 Natural Language Processing

Natural language processing is traditionally divided into three categories (Allen 1987):

syntax the relations between words, including grammatical structure;

semantics the meaning of words and how they combine to give sentences meaning;

pragmatics the meaning of words and sentences in a specific context.[1]

Understanding language and generating language are the two major problems of computational linguistics. The natural language understanding problem is to take a sentence and map it into some semantic representation that allows a computer system to reason about what was said in the sentence. This usually involves parsing, a process which creates a parse-tree describing the grammatical function of each part of the sentence. Parsing is a first check at whether the sentence is sensible or not, and is also a first step towards a more complete understanding of the sentence's meaning. A major difficulty in parsing natural language is dealing with the wide variety of words that appear in everyday language. Parsers need both extensive lexicons, including immense amounts of word-level knowledge for recognizing names, dates, titles, newly coined words, para-language, etc. Allen (1987) is a good general text on natural language processing; Winograd (1983) provides extensive coverage of natural language parsing, and Hirst (1987) is concerned with semantic interpretation (which typically occurs after parsing).

Natural language analysis is an important component of any natural language dialog system because what to say at a given time will depend on what has previously been said. In this section, we chronicle some of the work done in language analysis and discourse processing, and then discuss natural language generation in Section 7.1.1, in order to contrast the three-part model of turn-taking with previous work in generation.

[1] In classical linguistics, the lexical, morphological, and phonological domains are the traditional categories. The categories given in (Allen 1987) are more common in sentence-based computational linguistics.

2.2.1 Dialog

Dialog usually falls under the heading of pragmatics, and it includes both natural language understanding and generation.[2] While theories of discourse might be inspired by natural language, or is usable within a natural language environment, there is often no *necessary* connection between the two, meaning that many multi-agent interaction situations can be modelled as discourse situations. From the point of view of computer science, perhaps the most intriguing non-linguistic domain of discourse is human-computer interaction. In a graphical user interface, for example, the system/user interaction can be thought of as a conversation, where the user "utterances" consisting of mouse-actions and typing, and the system utterances consist of screen-changing actions, such as opening windows, displaying buttons, changing colors, etc.

Grosz and Sidner's Theory of Discourse

Grosz and Sidner (1986) describe a significant theory of discourse that has become one of the standard models for computational discourse. They propose that a discourse consists of three main parts:

- the *linguistic* structure divides a discourse into contiguous spans of text;
- the *attentional* structure concerns the *focus* of the discourse, and, for example, is used for determining the referents of pronouns and possible coherent topic shifts;
- the *intentional* structure encodes the purposes the speakers intend for each segment to have, and this typically involves using logic to describe the relevant mental attitudes of the conversants.

[2]The terms *dialog*, *discourse*, and *conversation* will be used more or less synonymously in this thesis. Discourse typically includes non-dialog texts generated by a single agent, although we will usually only be concerned with two-agent dialog-style discourse.

The linguistic structure of the discourse is important because it divides a discourse into semantic units, somewhat like scenes in a play or movie. For example, an essay[3] might talk about a boat in a segment that spans five paragraphs. The first three paragraphs might discuss the interior of the boat, thus forming their own subsegment, and the last two might talk about the exterior of the boat, forming another subsegment. Within each of these subsegments, there can be further segments that talk about more specific aspects of the interior/exterior of the boat. Boundaries between segments are sometimes given explicitly, e.g. a book is divided into chapters, each of which are divided into sections, which contain individual paragraphs. However, for text within a paragraph, there are usually no explicit segmentation markers, and one must use linguistic and contextual knowledge to locate segment boundaries. Developing techniques for automatically segmenting text is a research problem of some interest (e.g. Passonneau and Litman 1993; Hirschberg and Litman 1993).

Discourse segments are a basic unit of discourse. Each segment has a purpose, and is related hierarchically to other segments. For each segment, Grosz and Sidner require that one main purpose be associated with that segment, called its *discourse segment purpose* (DSP). Purposes are described in terms of intentions. For example, by giving you a five-paragraph description of my boat, I may intend for you to believe that I am rich, and this intention is the DSP of the segment corresponding to the entire five-paragraph text. The DSP of the segment that describes the interior is "I intend for you to believe that the interior is opulent", and the DSP of the segment describing the exterior is "I intend for you to believe that the exterior of my boat is hand-crafted". Taken together, the DSPs of these two subsegments are meant to support the overall DSP, and Grosz and Sidner would say that the subsegment DSPs *contribute* to the satisfaction of the main segment

[3]Grosz and Sidner's theory of discourse is meant mainly for multi-party discourse, but it can be used for describing ordinary written text. We do so in this case just for simplicity.

DSP (inversely, the main segment DSP *dominates* the two subsegment DSPs). DSPs can also be related by *satisfaction precedence*, which requires that one intention be satisfied before another. For example, the two subsegment DSPs in the boat example should be satisfied before the main segment DSP.

These two relations, dominance and satisfaction precedence, are the only DSP segment relations. This contrasts with a number of other theories of discourse structure, which stipulate numerous segment relations. For example, Rhetorical Structure Theory (RST) (Mann and Thompson 1988) includes textual relations such as *justifies*, *elaborates*, *sequence*, *volitional-cause*, *restatement*, *summary*, etc. In Grosz and Sidner's theory, the equivalent aspect of these relations are encoded in the DSP intentions, and describe changes that the participants would like to bring about in the mental state of the participants. In contrast, RST (and similar theories) hard-code the set of actions that participants can attempt, so there is no need to refer directly to individual mental states the way Grosz and Sidner do. The advantage of Grosz and Sidner's perspective is that it is much more flexible in the kinds of actions it allows the participants to bring about, i.e. they can attempt to influence someone else's mental states in any way that they can describe. RST limits one to the actions encoded by the given list of about 25 different text relations, but has the advantage of being much higher-level and closer to the surface-structure of the text. Grosz and Sidner do not attempt to give a list of sensible relations like those proposed by RST, and instead provide a belief-level language of intentions flexible enough to include RST relations and more.[4] This suggests that it should be possible to essentially decompose RST-like relations into Grosz and Sidner-like

[4]It is possible to add new relations to RST; Mann and Thompson's justification for their original set of relations is based on careful linguistic analysis of numerous texts. Presumably, to encode the equivalent relations in Grosz and Sidner's theory, a similar justification would be necessary. While it is easy to assign DSPs, it also easy to make up new RST-like relations, but whether or not these are justified is an independent issue.

DSPs; such multi-level theories of discourse have been suggested before in the context of planning e.g. (Carberry 1990; Litman and Allen 1990). However, it is perhaps unfair to compare the two models too closely, since they were designed to meet different goals: RST was designed as a practical tool for natural language generation, while Grosz and Sidner wanted a complete theoretical description of discourse.

Many discourse segments begin with a special *cue* (or *clue*) phrases that indicates its purposes. For example, a segment that contains an example might begin with the cue phrase *for example*. Cue phrases, such as *in contrast*, *on the other hand*, *therefore*, *and*, *but*, *thus*, *furthermore*, etc., can, in the right circumstances, be read as indicators of the type of relation between discourse segments. For instance, Cohen (1987) uses cue phrases to analyze the structure of argumentative discourse; Hirschberg and Litman (1993) discusses cue phrase disambiguation using prosodic information, plus provides a good general overview of the topic.

The attentional structure in Grosz and Sidner's theory tracks what is in focus in the discourse:

> The attentional state is property of the discourse itself, not of the discourse participants. It is inherently dynamic, recording the objects, properties, and relations that are salient at each point in the discourse. The attentional state is modeled by a set of focus spaces; changes in attentional state are modeled by a set of transition rules that specify the conditions for adding and deleting spaces. We call the collection of focus spaces available at any one time the *focusing structure* and the process of manipulating spaces *focusing*. (Grosz and Sidner 1986, p.179)

Each discourse segment has one focus space containing the most salient entities for that segment. An entity is salient if it is explicitly mentioned, or has been added in the

processing of that segment. A stack is used to order the focus spaces according to the dominance relations between segments.[5] The entities in a focus space determine the candidates for referring expressions and other anaphora in the current discourse.

Grosz and Sidner's model is a relatively comprehensive theory of discourse suitable for computers. It brings together a number of different aspects of discourse, and shows how they depend upon and relate to each other. In many ways, it is the de facto standard theory of discourse in computational linguistics because it synthesizes a number of pieces of work in a coherent and general framework.

2.3 Doing Stuff with Language

In a series of twelve lectures, Austin (1975) introduced the idea of *speech actions* (or just *speech acts*), which forms a foundation for much work in computational discourse. Speech act theory was introduced into AI by Cohen and Perrault (1979), and it fits well with traditional AI planning and plan recognition problems. In speech act theory, people communicate by performing speech acts, which are actions just like kicking a ball or lifting a box. We will briefly discuss speech acts, emphasizing the difference between them and the more typical physical actions that AI systems usually deal with.

2.3.1 Speech Acts and Planning

Speech acts are traditionally divided into three types:

locutionary act the action of a speaker uttering a sentence, corresponding roughly to the physical act of making sounds, gestures, etc.;

[5] McCoy and Cheng (1991) generalize the concept of a focus *stack* to a focus *tree*, and Hovy and McCoy (1989) show how focus trees can help constrain natural language generation.

illocutionary act the action performed by a speaker, such as questioning, warning, ordering, informing, etc.; utterances have corresponding illocutionary-act types, such as a warning, a question, an order, an informing, an insult, a joke, etc.;

perlocutionary act the action brought about by an utterance, such as convincing, persuading, deterring, surprising, etc.

Most work on speech acts is concerned with illocutionary and perlocutionary acts, since that is where content and meaning come into play. In computational discourse theory, we typically imagine a speaker and a hearer having a conversation. The speaker says things, and roughly the intent of those utterances, i.e. whether they are questions, answers, informs, clarifications, etc., are illocutionary acts. The effect that the illocutionary act has on the hearer is the perlocutionary act. A speaker usually wants to achieve a particular perlocutionary effect, but it is not entirely up to the speaker to determine if their intended acts achieve their intended effects. For example, suppose A and B are debating the merits of tele-learning in high schools, and A wants to convince B (i.e. achieve the perlocutionary action of convincing B) that the necessary video equipment is inexpensive. To achieve this convincing, A performs an inform action by asserting *It will only cost $15,000 a year to maintain the hardware.* It might happen that instead of B being convinced that tele-learning is cheap, B is in fact convinced that tele-learning is too expensive to use. Thus, the perlocutionary effect achieved is that B is convinced of the high cost of tele-learning, while A's intended effect failed. As another example, suppose Madge says to Trygve *Watch out for that snake.* Madge has performed a warning action, and hopes to achieve the perlocutionary effect of warning Trygve about the snake. Trygve could take Madge's utterance to be a warning, or, in some circumstances, he could take it as an insult (since he is perhaps well-aware of the danger of the snake), even while recognizing that Madge intended it to be a warning.

Traditional AI planning (discussed more in Chapter 4) is the problem of finding a sequence of actions to achieve a given goal. Actions are typically described in terms of pre-conditions and effects: for an action to be possible, its pre-conditions must hold, and, if an action is successfully executed, its effects are guaranteed to hold. For example, imagine going on a trip to England, which requires both buying a plane ticket and flying to England. The BUY-TICKET action is defined as:

action BUY-TICKET
pre: have money, travel agent available
effect: have ticket

This means that the BUY-TICKET action is possible only when the agent has money, and there is a travel agent available to purchase a ticket from. If those pre-conditions hold, then the agent can perform the action if it so desires, which will cause the agent to have the tickets.[6] We define the FLY-TO-ENGLAND action as follows:

action FLY-TO-ENGLAND
pre: have ticket
effect: in England

A typical planning problem would be to find the sequence of actions that achieves the effect of being "in England" (assuming the agent starts elsewhere). In this example, the agent must purchase a ticket before being able to fly to England, so BUY-TICKET must come before FLY-TO-ENGLAND. To buy a ticket, an agent needs money and the services of a travel agent, so earlier actions will be necessary, e.g. he might first save a few dollars every week until he has enough money to buy a ticket, and then phone the travel agent to find out when they are open for business, etc.

[6]There are many subtle details of representation and planning in general that we are ignoring in this example; Chapter 4 elaborates some of these issues.

Using the theory of speech acts, generating a conversation can be turned into a planning problem (Cohen and Perrault 1979), where the agent must decide what illocutionary acts (i.e. questions, informs, requests, clarifications, etc.) it wants to perform based on the (perlocutionary) effects it wants to achieve. If a set of speech actions can be defined in terms of pre-conditions and effects as in the travelling to England example, then standard AI planners can help a conversational agent to both understand what a speaker says, and to determine how it should respond (e.g. Cohen and Perrault 1979; Carberry 1990; Litman and Allen 1990). Furthermore, it is possible to mix speech acts with other kinds of actions, since speaking is often one way to get things done. For example, instead of saving money to buy a plane ticket to England, an agent can try to get money saying "Your money or your life!" to someone, in hopes of achieving the perlocutionary effect of having the victim hand over all their money. The conditions under which such a tactic will work matter greatly, and are the subject of basic research in speech act theory (e.g. Searle 1970). For example, it would help if the agent were dressed as a robber, and pointed a gun at the victim while speaking. In the wrong circumstances, the threat might be taken as a joke or an idle threat by the intended victim.

2.3.2 Plan Recognition

Dave: Open the pod bay doors, HAL.

HAL: I'm sorry Dave, I'm afraid I can't do that.

Dave: What's the problem?

HAL: I think you know what the problem is just as well as I do.

Dave: What are you talking about, HAL?

HAL: This mission is too important for me to allow you to jeopardize it.

Dave: I don't know what you're talking about, Hal?

HAL: I know you and Frank were planning to disconnect me, and I'm afraid that's something I cannot allow to happen.

2001, A Space Odyssey

The idea that speaking may be part of an agent's larger plan is an important one. Agents often need information to achieve their goals, and sometimes the best course of action is to ask a question of another agent who you think can supply you with the information. Suppose A asks B *Where can I get a picture taken?*. If B knows that A is in the process of planning a trip to England, then B could reasonably infer that A wants to get a passport photo, instead of, say, a family portrait, and so respond more appropriately. Even if B were uncertain of A's plans, B might find out what kind of picture A wants to get, in order to avoid, for example, wrongly directing A across town to the city's only family portrait studio. In this example, in order to respond helpfully, B is reasoning about A's goals and plans. In computational discourse, this is usually treated as a *plan recognition* problem. Plan recognition is the problem of inferring an agent's plans and goals by observing their actions. If you can infer part of someone's plan, this can often help you better understand their actions, and be more cooperative when dealing with them (Joshi et al. 1984).

Carberry's TRACK System

To give the reader a feel for a typical plan-based dialog system, this section will look briefly at the work of Carberry (1990), who developed a detailed model of natural language discourse plan recognition. She is interested in information-seeking dialogs where the information-seeker is trying to construct a task plan to be executed after the dialog terminates. The assumption of a task-oriented dialog is a common one, and it covers a wide range of interactive situations. Her model distinguishes between three kinds of plans:[7]

task plan a plan that achieves a task-related goal, such as being in England by earning money, buying a ticket, and then flying to England;

problem-solving metaplan this is a plan for how to construct a task-related plan, and could encode strategies like "divide and conquer" or "before worrying about anything else, make sure you have enough money in your bank account to purchase a plane ticket";

communicative plan a plan for making an utterance that achieves some particular intended effect on the listener.

Carberry uses a very descriptive planning formalism; for example, plans/actions have *applicability conditions* that determine when a plan is relevant, there are pre-conditions that determine when a plan can be executed, and the model distinguishes between primary (i.e. intended) and secondary effects of actions. Figure 2.1 shows a block diagram of Carberry's plan inference system named TRACK. Phase 1 takes a semantic representation of the user's utterance and hypothesizes a set of goals and plans that the user *might* currently be focusing on. In this course advising domain, the semantic representation of

[7] This tripartite model of dialogue was proposed by Lambert and Carberry (1991).

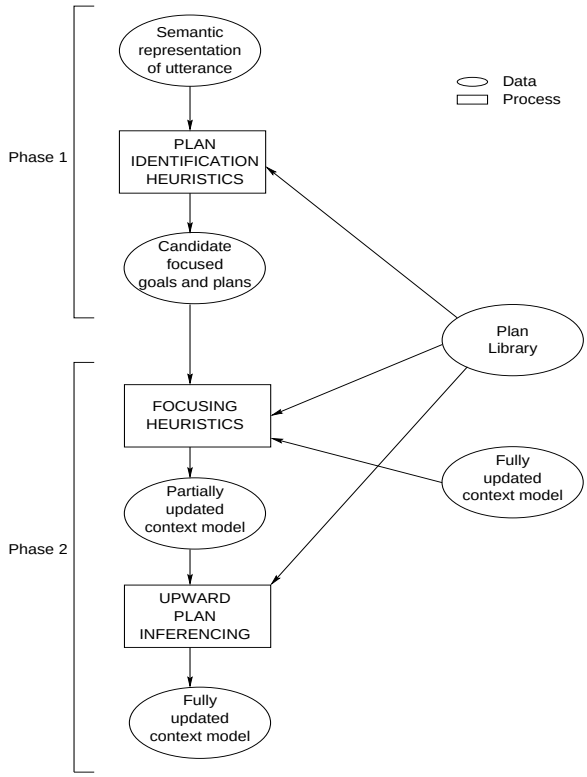


Figure 2.1: Block diagram of Carberry's system, based on Figure 3.1 of Carberry (1990).

the query "Is French 112 a three credit-hour course?" is

$$\text{Request}(S, H, \text{Informif}(H, S, \text{Credits-of}(\textit{FRENCH112}, 3)))$$

S and H are the speaker and hearer. The predicates are interpreted intuitively, and this expression says that S has requested that H inform S if French 112 has three credit-hours. From this it is deduced that the speaker wants to know if French 112 has three credit hours:

$$\text{Want}(S, \text{Knowif}(S, \text{Credits-of}(\textit{FRENCH112}, 3)))$$

Note that concepts such as course and credit-hour are compiled into the domain model, so such a system is not likely able to answer questions such as "What is a credit-hour?" without extra facts for answering such questions put into its database.

Eight domain independent heuristics are used to associate these hypotheses with plans in the system's domain-specific plan library. An example of such a rule is:

> *Rule P4:* If the information-seeker wants to know how to achieve a goal or whether he can achieve a goal, that goal becomes a candidate focused goal and those plans that contain the goal on their primary effect list become associated candidate focused plans.

The concept of focusing is important in Carberry's work, and is used to cut down on the possibilities that must be considered during the reasoning process.

The output of phase 1, a set of *candidate focused goals* and *candidate focused plans*, is fed into phase 2 which is responsible for inserting its input into the global model of the conversation. Tree-structured *context models* are used to represent the system's beliefs about the inferred plans and goals. The nodes of a context model are plan/goal pairs (the goal being what is achieved by the plan), and a child node's goal is a pre-condition or sub-goal of the plan associated with its parent. The candidate plan/goal pairs from phase 1 are inserted into the context model using focusing heuristics and by monitoring the global context, which is considered to be the path of nodes from the context model root to the current goal node in focus. The subplan believed to be most appropriate is then selected and inserted into the context model to become the new focus of attention. Carberry (1990, chapter 3) gives more details and examples of the use of context models for representing the system's beliefs. Carberry then applies her model of discourse to the problem of handling a particular class of pragmatically ill-formed utterances, and inter-sentential ellipsis. An example of inter-sentential ellipsis can be seen in this dialog:

Dialog 2.1

Student: Is CS440 being taught next semester?

Advisor: Yes.

Student: Any sections at night?

In the last utterance, the advisor must infer that the student is referring to sections of CS440; Carberry's approach provides a systematic way of handling a number of different types of this phenomena.

2.3.3 Limitations of the Speech Act Approach

The marriage of speech act theory and AI has been a fruitful one; however discourse still poses problems not addressed by the speech-act framework. Here we give a list of some of the concerns left unresolved by the speech act approach to language:

- Traditional deliberative planning, even in its simplest form, is a computationally difficult problem, but it only gets more difficult due to uncertain or incorrect knowledge of the world, non-deterministic actions, concurrent actions, or a dynamically changing world; all of these issues, and more, show up in real-world discourse;
- Traditional planning systems are usually non-incremental and assume a static, unchanging environment, e.g. they fix the set of possible actions. Logical modelling makes *hard* constraints (see Section 4.3.1) easy to deal with, and often domain models make idealizing assumptions that turn what are soft-constraints in the real world into hard constraints in the domain model. For instance, in the standard blocks-world domain (see Figure 4.1), the predicate $on(A, B)$ is used to describe block A sitting on top of block B. When you stack blocks in the real world, it is possible to have cases where one block is sitting on top of another such that you

cannot place another block on it without toppling the structure; representing such variation effectively in logic is tricky since one must worry about the shape and mass of the blocks, and be able to distinguish between slight differences in position. Also, if A and B are the appropriate shapes, it might be possible to balance A against B such that A is touching the top of B and the table, which violates the common blocks world assumption that a block cannot be both on the table and on top of another block. This sort of idealization leads to some fundamental limitations on the kinds of discourse that plan-based dialog systems will be able to handle. For example:

- In the course advising domain, most courses have pre-requisites that a student must fulfill in order to be able to take the course; yet most logic-based systems do not allow for constraints to change, i.e. for pre-conditions for actions to change as the environment changes. Suppose the student has not taken, say, the MATH100 pre-requisite for a university math course, but has come from a reputable university having taken equivalent pre-requisites (or in some other way shows the necessary aptitude course). Sometimes, students can negotiate for exceptions to rules to be made, and so such cases show that the rules in a course-taking domain are not always hard constraints;
- Suppose A wants to go to England, but cannot afford to spend over $1500 on the trip. In terms of planning the trip, this is an important and useful hard constraint, i.e. a cost bound constrains almost every aspect of his trip, influencing where he can stay, what he will eat, and how he can travel. But suppose A is a soccer fan, and learns that a World Cup qualifying match will be played in England during his stay. To also attend this match, the whole trip will cost at least $2000, which is over the $1500 limit. However attending this

soccer match might be of such value to A that he considers new options that he would not otherwise consider, such as borrowing \$500, or taking a night-job to earn some extra money, or even dropping his initial goal of travelling around England in favour of traveling to see this World Cup qualifying match, which just happens to be in England.

These and similar examples suggest that discourse is better treated as a *dynamic* reasoning problem, where the structure of the problem can change, e.g. a hard constraint becomes a soft one. By basing a discourse system on standard AI planning, one is committed to a single problem P that cannot tolerate changes in the environment such as those in the above examples;

- Speech act theory does not treat people as *processing* agents, which they are, or as *resource bounded* agents, which they also are; for example, suppose you go to a restaurant[8] where they let you pick your own condiments from a display. The commonly understood names of these choices are: lettuce, onions, relish, mustard, peppers, tomatoes, mayonnaise, and ketchup. You simply order the condiments you want on your sandwich by listing them, e.g *I would like lettuce, onions, mayonnaise, and relish.* But some combinations of condiments can be described more succinctly than by listing their names. Descriptions like *everything*, or *everything but peppers* are common, because they are briefer than providing a list of all the desired condiments. A more extreme example would be a customer who orders *all condiments with double letters in their names*, instead of saying *lettuce, mayonnaise, and peppers.* The problem with this order is that it both breaks the standard restaurant schema for how condiments are ordered, plus it requires greater mental effort from the server to decode its meaning. This latter constraint is important, and suggests

[8]Say, the same one as in Section 6.4.3.

that one should take into account the processing effort involved in understanding utterances. This concern is partly implicit in systems that model a user's expertise, since if A asks B a technical question in an area unfamiliar to B, B is unlikely to be able to answer (or even understand) the question without a lot of effort on their part, e.g. looking in a reference book, getting more schooling, etc. Section 3.3 explores this idea in more detail.

These and similar examples essentially show that a computational theory of discourse must take into account the fact that conversants are resource-bounded agents. The processing effort must, at least sometimes, be taken into account, and this is usually not addressed by speech act theory, or the computational systems upon which they are based. Since most such systems are written-discourse systems, i.e. they assume that they will be hearing and generating written text as opposed to spoken language, it is reasonable to assume that the agents will have some extra time to revise what they want to say (just as a writer can take as much time as they want to write a sentence). However, for systems that are closer to spoken-discourse, processing effort must be taken into account, and the basic speech act approach to discourse does not cover this.

2.4 Oreström and Turn-taking in English Conversation

Many of the basic turn-taking ideas developed in this thesis are based on work done by Oreström (1983), a linguist who presents a theory of turn-taking based on the study of spontaneous English conversation. In this section, a summary of Oreström's work will be presented, emphasizing the parts most relevant to this thesis.

Oreström distinguishes between two kinds of utterances: a *speaking-turn*, and a *back-channel* item. Speaking turns typically convey new information and expand the topic of discussion. Back-channel utterances are "...certain brief, spontaneous reactions from

the listener" (p. 23). Oreström says:

> Examples of these listener responses are *m*, *mhm*, *yes*, *yeah*, *right*, *fine*, *OK*, *alright*, *I see*, *that's right*, etc. (other than answers to questions). They represent rather special functions where the listener informs the speaker that his message has been received, understood, agreed to and/or has caused a certain effect thereby supplying him with direct feedback. Such utterances are normally not, if ever, picked up and commented on by the other speaker but are still important contributions as they help to sustain the flow of interaction. (Oreström 1983, p.23)

This basic distinction in utterance types is captured in the turn-taking logic presented in Chapter 5; essentially, if a conversant without the floor (i.e. a listener) makes an utterance without attempting to take the floor, then that utterance is a back-channel utterance.

Figure 2.2 is a list of observed facts about turn-taking in human conversation. An important aspect of turn-taking is that turn shifts happen around *transition relevance points*, or TRPs (Sacks et al. 1978). Oreström's study found that the following five linguistic features have the highest correlation with TRPs:

prosody completion of a tonal unit with a non-level nucleus;

syntax completion of a syntactic sequence;

semantics completion of a semantic sequence;

loudness decrease in volume;

silent pause following immediately after the end of a tonal unit.

1. Speaker-change recurs, or at least occurs
2. Overwhelmingly, one party talks at a time
3. Occurrences of more than one speaker at a time are common, but brief
4. Transitions (from one turn to the next) with no gap and no overlap are common. Together with transitions characterized by slight gap or slight overlap, they make up the vast majority of transitions
5. Turn order is not fixed, but varies
6. Turn size is not fixed, but varies
7. Length of conversation is not specified in advance
8. Relative distribution of turns is not specified in advance
9. Number of parties can vary
10. Talk can be continuous or discontinuous
11. Turn-allocation techniques are obviously used. A current speaker may select a next speaker (as when he addresses a question to another party); or parties may self-select in starting to talk
12. Various 'turn-constructional units' are employed; e.g. turns can be projectedly 'one word long', or they can be sentential in length
13. Repair mechanisms exist for dealing with turn-taking errors and violations; e.g. if two parties find themselves talking at the same time, one of them will stop prematurely, thus repairing the trouble

Figure 2.2: Observable facts about turn-taking (reported in Oreström 1983).

The first three factors form a *grammatical boundary*, and according to Oreström represent a "major juncture" in a discourse. The more such features that appear together, the more likely that the point in question is a TRP.

Conversants coordinate their interaction using *turn-taking signals*. Oreström reports the following four kinds of turn-taking signals that appear in human conversation:

turn-yielding signal the speaker indicates that he intends to terminate his turn

attempt-suppressing signal the speaker indicates that he attempts to hold the turn

turn-claiming signal the listener indicates that he attempts to take the turn

back-channel item the listener indicates that he does not intend to take the turn

Oreström studied numerous aspects of turn-taking in English conversation; some of his particular findings are listed in Figure 2.3. His work was not computational in nature, although much of what he presents is amenable to computerization, and so it forms a good starting-point for the turn-taking ideas in this thesis, most notably in Chapter 5, where turn-taking signals are formally defined, along with offers and requests for the conversational floor.

2.5 Speech Processing

While this thesis does not deal directly with speech processing, it does promote a move from written-discourse to spoken-discourse systems, and suggests adding turn-taking to written-discourse systems as a first step. Here, some of the main issues in speech processing will be summarized, based mainly on papers from the collections (Waibel and Fu Lee 1990) and (Ramachandran and Mammone 1995), to give some idea about the issues relevant to spoken dialog systems.

Speech processing has two major divisions, speech recognition and speech synthesis (generation), similar to the distinction between parsing and generation in natural language processing. A speech recognizer transforms sounds (e.g. represented as a waveform) into words, and from there natural language understanding techniques can be used to convert the words into a suitable semantic representation. Young et al. (1990) list

- 95% of speaker-shifts (in cases where there was no over-lapping speech) occurred when three points are simultaneously reached (i.e. at a grammatical boundary [a GB]):

  1. the end of a tone unit;
  2. the end of a syntactically complete sequence;
  3. the end of a semantically fully rational sequence.

  Two other linguistic features correlate with turn-taking: loudness (i.e. a decrease in volume), and silent pauses at the end of tone units;

- 75% of GBs occur in combination with reduced loudness in the tone unit just before the GB; this acts as a *pre-signal* to the listener that the next GB might be a speaker-shift point;

- There are no specific and unambiguous markers of turn-finality, therefore it is better to talk about "signals of possible turn-yielding" than "turn-yielding signals";

- Turn lengths (in terms of words) in conversation tend to be relatively short, e.g. only 5.2% of turns were longer than 100 words;

Figure 2.3: Some of Oreström's findings on turn-taking in English.

some of the major problems that arise in speech recognition but not in parsing written texts:

- There is uncertainty in what a person has said (and there is no uncertainty in what a person types);

- Word boundaries have no explicit markers in speech (like spaces in written text);

- Speech can suffer from phonetic ambiguity, e.g. "I scream" and "ice cream" sound the same;

- Spoken language is often terse and syllables can be omitted, e.g. "United States" might be pronounced like "unite states";

- Spontaneous speech is often more ungrammatical than written language.

These and related problems are what make speech recognition a distinct subfield from natural language processing. A number of techniques have been applied to speech recognition, for example stochastic methods (e.g. Markov models), neural networks, high-level knowledge techniques, and template-based approaches (Waibel and Fu Lee 1990; Ramachandran and Mammone 1995).

Reddy (1990) is an early but still useful review of the basic issues in speech recognition. He categorizes the different variations of the speech recognition problem along these dimensions:

Mode of Speech Isolated words or connected speech? Isolated word speech recognition is a much easier problem than connected speech recognition, but requires the user to pause between each word when speaking, which is unnatural for many people, although it can be learned with practice. Connected speech is ordinary speech, with no special pauses between words. This turns out to be a more difficult problem since the system must locate and mark word boundaries;

Vocabulary Size The size of the understood vocabulary makes a difference. It is easier to recognize words from a small vocabulary, as compared to a vocabulary of hundreds, thousands or even an unlimited number of words;

Task Specific Information As in natural language processing, addressing a specific task can greatly reduce the number of possible interpretations of an utterance;

Language Restricted command-languages, or English-like languages, can be quite helpful to a speech recognition system. For instance, in an artificial language phonetic ambiguity can be controlled or eliminated completely;

Speaker Speakers can be cooperative or not uncooperative. For example, someone using a speech-based word processing system might be cooperative in the sense that they speak carefully, work in a room with little noise, and tolerate (and expect) mistakes from the recognizer. A not uncooperative speaker is not trying to confound the system, they are just not necessarily giving the recognizer any special consideration. For example, a passenger rushing for a train could ask an automated information kiosk which way to go, with no extraordinary concern for the ambient noise, the grammaticality of their utterance, the fact that they are out of breath, etc.;

Environment Is the environment noisy or quiet? Speech recognition in a quiet room is much easier than speech recognition in a noisy room.

Reddy gives a continuum of six kinds of speech recognition systems that vary along the above dimensions. At the low end are "isolated word" systems that only need to find individual words from a small vocabulary; at the high end are "unrestricted connected" speech recognition systems, which must contend with unrestricted speech, no task assumptions, a not uncooperative speaker, and a quiet room.[9]

Speech generation is the problem of translating strings of natural language text into speech (O'Shaughnessy 1995). Simple real-time text-to-speech translation is currently possible, but there is still much room for improvement in intelligibility and naturalness of the generated speech. In practice, there is not the same pressure to create general speech generators as there is to create speech recognizers, because pre-recorded speech suffices for many applications, just as pre-written text suffices for many systems that display language.

Spoken dialog systems combine speech recognizers and generators such that humans

[9]Obviously, an unquiet room would add another level of complexity, but the described system is already a very tall order, and in practice headset microphones can be used which can help create a quiet environment.

and computers can engage in spoken conversation. Given the tremendous technical difficulties in getting these parts to work in isolation, it perhaps unsurprising that comparatively little research has been done on spoken dialog. As suggested above, the speech recognition part appears to present the most immediate practical difficulties, and so dialog systems are typically most limited by the ability of their recognizers. But even with perfect speech recognition and generation, dialog systems must still worry about understanding, which has been the major focus of much work in natural language understanding. The turn-taking aspects of most dialog systems are about as simple as in their text-based counter-parts, e.g. the MIND system (Young et al. 1990), the dialog system of Smith et al. (1995; see Section 7.6), and the TRAINS system (Allen et al. 1995) all use simple, regimented turn-taking that allows no mid-turn interruptions. More recently, Walker et al. (1997) do consider the possibility of mid-turn interruptions. However, most of this work is still at the early stages, and is being mentioned as a natural direction for further research.

Chapter 3

Architecture for a Turn-taking Agent

> "Albert," I said, "tell me something. You computers are supposed to be lightning-fast. Why is that you take so long to answer sometimes? Just dramatic effect?"
> "Well, Bob, sometimes it is," he said after a moment, "like that time. But I am not sure you understand how difficult it is for me to 'chat.' If you want information about, say, black holes, I have no trouble producing it for you. Six million bits a second, if you like. But to put it in terms you can understand, above all to put it in the form of conversation, involves more than accessing the storage. I have to do word-searches through literature and taped-conversations. I have to map analogies and metaphors against your own mind-sets. I have to meet such strictures as are imposed by your defined normatives for my behaviour, and by relevance to the tone of the particular chat. 'Tain't easy, Robin."
>
> *Gateway, Frederick Pohl*

3.1 Introduction

This chapter outlines the general model of turn-taking, which is then expanded upon in Chapter 4 and Chapter 5. It also provides a detailed overview of the initial motivation step. We begin with a brief discussion of turn-taking as a phenomenon of general multi-agent interaction.

3.2 Turn-taking as a General Feature of Interaction

Turn-taking is not just a language phenomenon. Essentially, in any situation where multiple agents must share a single indivisible object, some kind of turn-taking will occur with that object. For example, in baseball there is a single ball that usually only one player can possess at once. More abstractly, in a hierarchical decision-making structure, as found in companies, often only one agent (e.g. a manager or a committee) holds the final responsibility for making certain kinds of decisions. Over time, different agents will be made responsible for these decisions, and this can be seen as agents taking turns possessing the decision-making responsibility.[1]

We will usually only be concerned with turn-taking as it occurs in dialog, although it is interesting to note that while taking turns is necessary for conversation, it is not sufficient.[2] Consider a hockey broadcasting team, which consists of at least a play-by-play announcer and an analyst. The announcer speaks most often during the game, describing the action. During stoppages in play, or sometimes even during the play, the analyst will take a turn speaking, typically providing some sort of fact or insight beyond a description of what is happening. When the analyst speaks, the announcer is silent, and vice-versa. The two are not having a conversation with each other, but they are clearly taking turns speaking.

While the turn-taking model developed in this thesis is geared towards dialog, it is

[1] As an extreme example of the generality of turn-taking, one can imagine physical locations taking turns possessing objects. A basic fact of nature is that two solid objects cannot share the same location at the same time. Usually, one thinks of a location as being a property of an object, but we can turn this around and imagine that locations are able to possess at most one object at a time. This does not seem to be a practical way of thinking about the physical world, because it discretizes the notion of location, which seems quite unnatural. It might be a more useful perspective for discrete models of the world, e.g. the kind that are often used in AI, but we will not pursue this perspective any further.

[2] While, in general, people cannot and do not speak simultaneously, they may *sing* at the same time, usually to achieve harmonic effects not otherwise possible. Opera, for example, often has people singing different parts at the same time.

presented in sufficient generality to admit application to the other domains of interest. Any multi-agent system is a candidate for turn-taking, especially systems that can be seen as implementing a kind of conversation between the agents.

3.3 Three Steps to Taking a Turn

Our goal-oriented model of turn-taking explicitly addresses three main concerns: *why* to take a turn (motivation), *what* to say during a turn (which goal to address in the dialog), and *when* to take a turn (at relevant points in the dialog). Figure 3.1 shows the architecture of a turn-taking agent in its most general form. Each box represents a module of the turn-taking agent that operates in parallel. The agent is continuously receiving input from the environment, and raw perceptions are converted into goals as soon as they arise. Whenever new goals arise they are put in the working memory of step 2, and the agent continually reasons about which of the goals currently present it should attempt to achieve next. The turn-execution module is responsible for deciding when to execute the goal selected by the second step, or, if the agent is forced to act, it is responsible for performing an appropriate stalling action to indicate to the other agents why it is not yet ready to act. This whole process continues indefinitely.

The rest of this chapter contains an overview of the whole model, while Chapter 4 and Chapter 5 will expand on the last two of the three main steps. The motivation step is discussed next in Section 3.4.

3.4 Motivation: Why Should I Take a Turn?

The motivation for speaking in a dialog is often straightforward, e.g. if someone asks you a question, then you ought to answer it (all other things being equal). In computational discourse, dialogs are often assumed to be task-oriented, and thus they are a communica-

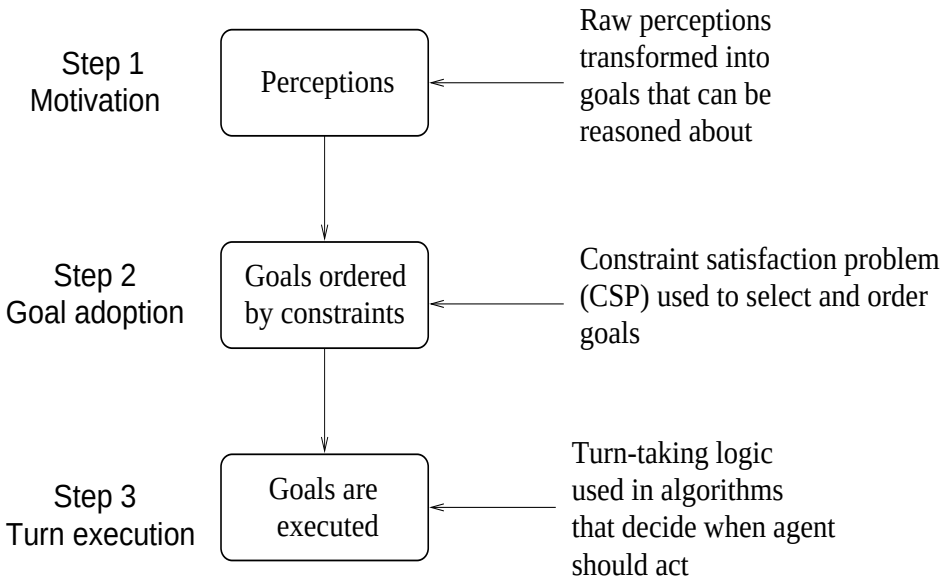


Figure 3.1: The 3-step model for taking a turn.

tion tool used by two or more agents committed to achieving some domain goal. In such cases, agents speak because they need information, want to coordinate their activities, must clarify ambiguities, want to clear-up confusion, etc. The nature of the task is a major force in shaping the dialog, and often in determining what agents will say and when they will speak (see Section 2.2.1). Task-oriented dialogs are of practical interest since many activities are naturally modelled as achieving some task, e.g. creating a schedule of courses, preparing a meal, getting money from a banking machine, writing an essay on a word processor, etc.

However, human conversation is not always strictly task-oriented. People talk to each other for many reasons: they might speak just to be friendly, to prevent boredom, to show-off (e.g. make a witty comment), or to entertain others. These kinds of conversation have goals, but typically not the same concrete sort as problem-solving dialogs. For example, two friends might chat about whatever catches their fancy or seems interesting. The

conversation serves the purpose of passing the time and preventing boredom, but there is no necessary set of topics that must be discussed to achieve this goal.

In practice, we will usually only be interested in dialogs that involve solving some well-defined problem. Motivations for speaking are then related to this task. For example, if one thinks of a dialog system as a theorem-prover, as in (Smith et al. 1995) (discussed further in Section 7.6), then the system has the well-defined goal of trying to prove its theorems (which come from the domain model). Roughly, such a system will talk to the user when it believes it lacks the necessary axioms to prove its theorems, or when one of the goals it has reached requires it to speak. One can imagine the internal reasoning of such an agent going like this:

> I want to prove G, and I know facts A and B. But using just A and B I can't prove G; but, I see that if I knew the truth-value of C I could prove G. So I should ask the user if he knows if C is true or false.

This provides a motivation for speaking (i.e. the system wants information to help prove a theorem), but it does not give any indication *when* the system should ask about C; this is discussed in Section 3.6. Furthermore, while Smith et al. (1995) certainly do give specific details about how reasoning ought to proceed, we use a different constraint-based reasoning formalism, discussed in Section 3.5, more appropriate for time-constrained reasoning.

Internal Goals and External Goals

When dealing with interacting agents, we can distinguish between two basic goal types: *internal* goals and *external* goals. Internal goals are goals that an agent tries to achieve on its own without consulting any external agent or source of information, and external goals are ones that an agent tries to achieve by consulting with some other agent or

information source. This distinction captures the fundamental idea that communication and reasoning are interchangeable, i.e. one can try to solve a problem by thinking about it alone, or one can do no thought and ask another agent if they know the answer; any combination of reasoning and communication between these two extremes is also possible.

Traditional AI problem-solving systems typically assume a single-agent that has internal access to all the knowledge necessary to solve a problem. If such an agent is able to prove it does not have the resources to solve a problem, then the problem is effectively unsolvable. In contrast, a communicative agent need not give up if it is unable to solve a problem itself. A communicative agent can ask other agents if they can solve, or help to solve, their problems. In the worst case, asking other agents for help will fail to solve a problem, but more heads are often better than one, and a group of agents might be able to collaborate on and solve a problem that none could solve alone. One can see that a communicative agent need not have much problem-specific expertise at all, and yet still be good at finding solutions to problems by knowing what other agents and knowledge sources to query.

This distinction between internal and external goals is an interesting and common one, and so we give a number of concrete examples:

- Bubblesort works by comparing elements of an array and swapping them if they are out of order. We can imagine replacing the comparison function with a query to the user, asking them to decide if x is less than y. Such queries would obviously be of little practical value for sorting an array of numbers or strings since a computer can more accurately and efficiently compare any two numbers. Possibly, for lists of objects for which the comparison function is incomplete, asking the user might be an option; for example, if required to sort a mixed list of numbers and strings, and not possessing a function for comparing a string to a number, the system could

resort to asking the user to input such a function at the beginning (asking the user to make comparisons every time a string/number comparison arises could be unbearably tedious);

- A person may set themselves the goal of, say, buying a stereo. It is possible for the person to buy a stereo without ever directly communicating with anyone else about which stereo to buy, e.g. they just go to the store and buy the first stereo they see. However, essentially any phase of this deliberation could be replaced, or augmented by, discussion with someone else, such as a sales clerk or a friend. If I want to decide whether Panasonic is better than Sanyo, I can either decide for myself, or ask someone else for their opinions and experiences. What I actually decide could be based entirely on my internal reasoning, entirely on the opinion of the person(s) I asked, or on both;

- An agent might execute both the internal and external versions of a goal in order to ensure that they are both in agreement on certain facts. For example, I might internally deduce that I should mail a package today, instead of tomorrow, and verify my reasoning by asking you if you agree. Disagreement indicates a potential inconsistency, just as if I had used two different internal reasoning methods that came to different conclusions about when I should mail the package;

- An agent that knows a certain goal is only an external goal can avoid wasting time trying to satisfy such goals internally. For example, if Trygve knows that Madge is working alone in her study and not in contact with anyone else, and Trygve wants to know if she is making progress, he must ask her, since this cannot be known through any amount of internal reasoning alone. If Trygve knows that she has asked not to be disturbed for the next hour, then he knows that he should not bother trying to determine if she is making progress until he is able to ask her at the end of the

hour. Another famous example of this distinction occurs in the novel *The Hobbit*, where Bilbo Baggins asks Gollum "What have I got in my pocket?" in a riddle contest. This question does not have the kind of answer one can be expected to deduce internally! This is exactly the kind of question that someone should ask of Bilbo, not the other way around.

3.4.1 A Simple Model of Motivation

The internal/external goal distinction provides a general way of framing the motivation step of a turn-taking agent. A turn-taking agent is a communicative agent, and so it must decide if the goals it wants to achieve should be treated as internal or external goals. In this and the following section, we sketch a simple model of motivation that an agent can use to decide if a goal should be achieved externally, i.e. by querying some other agent.

We will assume that rules are used to trigger goals in a turn-taking agent, i.e. if a given set of facts is true, then a particular goal is instantiated or action attempted. For simplicity, we only consider the case of an agent interested in discovering the truth-value of a single proposition p. A number of natural and general-purpose rules can be written that describe motivations. For instance, if an agent needs to know the truth-value of p, but does not currently know it, then the agent can ask the other agent if it knows whether p is true or false. Let agent A represent the system, and agent B the user. The following two motivational rules can be encoded in A:

B may have info A wants If A wants to know the truth-value of p, and A does not believe B does not know p's truth-value, then A can ask B if B knows p's truth-value.

A has info B wants If A believes B wants to know the truth-value of p, and A believes it knows p's truth-value, then A can tell B the truth-value of p.

The first rule is used when A needs information it is unable to supply on its own. A need not be certain that B know p's truth-value, just that B definitely does not know it. Exactly when and how to ask this question is the job of later steps of the 3-step turn-taking model (Section 3.5 and Section 3.6).

The second rule could be used as a motivation for A informing B of p's truth-value. All that is required is that A come to believe that B wants p's truth-value, which could happen directly (e.g. B asks *Is p true or false?*), or indirectly because of something else B says. In some cases, this rule might be responsible for A interrupting B, e.g. in the middle of a sentence A might realize that B needs information that A can supply, and so A could interrupt and give it right away, or wait for B to give up the floor; this is the basic issue of turn-taking addressed by step three of the turn-taking model (Section 3.6).

Other basic motivational rules can be added. For instance, if one agent notices an inconsistency with respect to some proposition p, that is either A believes p and B believes $\neg p$, or A believes $\neg p$ and B believes p, then the agent who notices this ought to point out that this problem exists.[3] Just what the agent says depends on p and what relation it has to current conversation or domain of interest. For example, if A and B are cooking dinner for a dinner party, and A believes there will be no vegetarian guests but B believes there will be at least one vegetarian guest, then it is likely important to resolve this inconsistency since it influences what dishes should be served. In other cases, the inconsistency might not be so important, or it might not be one that is easily settled. For instance, A might believe that Ronald Reagan is the best president the USA has ever had, while B might believe this is not the case; or A might believe that the play they just saw was well-acted, while B believes it was poorly acted. In fact, in some situations realizing a difference of opinion might be a good reason to *not* mention a certain topic,

[3]These rules are similar to the interruption motivations given by Whittaker and Stenton (1988), discussed in Section 7.10.

since it is sensitive or will only cause "unproductive discussion". Clearly, the relevance of inconsistent beliefs is important in their resolution, and in deciding if they should, or can, be resolved at all.

In the *SG* course advising system described in Section 6.4, a major motivation for posting a goal is to fix an error, such as a spelling mistake or a request for information about a course that does not exist. In *SG* the need for dispensing with goals that have become irrelevant is also made clear, and Section 6.4 discusses one general way of dealing with goal deletion.

Cost-based Motivation

A more sophisticated model of motivation would take into account the cost of discovering the truth-value of p. Suppose agent A can, without resorting to querying another agent, determine the truth-value of p at a cost of $C_A(p)$. This cost is a measure of the necessary effort to discover p's truth-value, and could include the time needed to retrieve p, the likelihood of getting the correct truth-value, etc. If A cannot possibly figure out the truth-value of p on its own, then $C_A(p) = \infty$. Suppose that the cost of determining the truth-value of p by asking B is $C_{A \to B}(p)$, which is the sum of the overhead of the interaction plus the cost of B's retrieval of p, i.e. $C_{A \to B}(p) = C_B(p) + (\text{cost of interaction to } A)$. When A needs to know the truth-value of p, it can either spend time trying to derive the answer itself at a cost of $C_A(p)$, or it can ask B at a cost of $C_{A \to B}(p)$. By estimating values for $C_A(p)$ and $C_{A \to B}(p)$, A can decide when it will likely be less costly to ask B a question as opposed to trying to figure out the answer itself. Note that this choice assumes that A wants to minimize the overall effort, which includes A's effort, B's effort, and the communication overhead. If A was only concerned with minimizing $C_A(p)$, then it might do so by browbeating B into solving its problems, resulting in a very high value for $C_B(p)$, hence a high value for $C_{A \to B}$.

While the exact cost of a query will depend on the proposition being retrieved and the facts known by A and B, it is possible to distinguish some basic classes of propositions:

Knowable by all agents for any agent A, $C_A(p)$ is likely to be small for basic obvious facts about the world such as these:[4]

- $p =$"One equals two"
- $p =$"One plus one is two"
- $p =$"Water is wet"

Knowable by particular agents only for example, Madge is likely to know the truth-values of each of these propositions, so $C_{\text{Madge}}(p)$ is likely to be small:

- $p =$"My name is Madge"
- $p =$"I am female"
- $p =$"I am awake"
- $p =$"My mother's name is Effie"
- $p =$"I had fish for lunch"

Some "personal" facts may or may not be known or as obvious to people other than Madge, e.g. B might not know Madge's name, or might only be able to recall her name after some time spent recollecting.

Knowable by expert agents only some propositions require expert knowledge to know or to learn the truth-value of, so $C_A(p)$ is likely to be small if A is an expert with respect to p, and larger otherwise:

[4]Of course, there is no consensus on what counts as obvious. Facts such as $p =$"Cher was married to Sonny Bono" might be known to a class of North Americans in a certain age range, but perhaps unknown to most other people. No serious claim is being made about what sort of basic facts all people are likely to know, and in practice this can be restricted to basic facts of a particular domain model, e.g. one might assume in a course-advising setting that both a student and an advisor understand the concept of a "teacher".

- p ="Stalin had a brother named Vitaly"
- p ="The fourth digit after the decimal in the square root of 2 is 2"
- p ="Wiles proof of Fermat's last theorem is correct"

These are similar to personal facts in that one reason they can be difficult to know is that one lacks the proper knowledge to determine their truth-value.

Unknowable by any agent for any agent A, $C_A(p)$ is likely to be very high:[5]

- p ="While in a coma just before she passed away, Marie imagined Pierre in his Sunday best"
- p ="Shakespeare swatted and killed exactly fifteen hundred flies during his life"

Such a classification can be used to "tag" propositions in a knowledge base in a way that can help an agent decide if it should treat a goal as internal or external. For example, if p is the sort of proposition only an expert would know the truth-value of, then an agent ought to try to find an expert who might know p, or, possibly, try to gain the expertise needed to know p (e.g. by reading books, taking a course, doing research, etc.) Or, if p is a personal fact about some agent A, then one might need to find and ask A directly, or ask a close friend of A. Also, such a classification can help an agent when planning. For example, if an agent starts a project and believes that the decisions that will be made along the way will consist mainly of personal and world facts it is likely to know, then it can expect it will not to need to query other agents. If it did expect to be faced with

[5]If one is willing to go into the abstract world of mathematics, more kinds of difficult to know, or even unknowable, propositions arise. For example, there are many open problems in mathematics, such as "Does $P = NP$?" which no one is sure of the truth value of. There are also classes of undecidable problems for which there is known to be no general algorithm for answering, e.g. determining if two context-free grammars generate the same set of strings.

a decision requiring outside expertise, then it could adjust its plans to make sure that it can communicate with the needed agents when the relevant decisions arise.

A very simple classification is used in the course-advising domain described in Chapter 6. There, an advisor and a student collaborate to produce a schedule of courses for the student. The advisor is assumed to be an expert on the course domain, and knows all the relevant information for creating a course schedule. The student does not have the domain expertise of the advisor, but the student does know her own preferences for the sorts of courses she wants to take. Thus a course schedule is constructed from hard domain constraints that the advisor knows about (e.g. what courses are offered, when they are offered, who is teaching them, etc.), and student preferences, such as the course topics or teachers they would most like to have. Usually, ahead of time, the advisor will not know the student's preferences, and the student will not know all the relevant facts about the courses, so the two must communicate to share their knowledge. Knowing about this difference between domain knowledge and student preferences ahead of time allows one to write algorithms that can make decisions based on this distinction, e.g. if the course advisor needs to know the truth-value of p, and p is a student preference, then the advisor knows that it will almost certainly need to ask the student about p. Similarly, not knowing if p is a preference or domain fact means that the advisor cannot be certain if the student must be questioned to determine the value of p.

3.5 Goal Adoption: What Should I do When I Want to Take a Turn?

The previous step is responsible for instantiating goals; this second step is responsible for selecting which goal(s) to pursue. In any complex domain, there will likely be many goals posted with no guarantee of any nice relations between them, e.g. goals may be

redundant or even inconsistent. Motivations are not sufficient for modelling turn-taking because simply wanting to say something does not mean you should say it. An agent will typically have more than one goal in mind at any one time, so it must be able to decide which one to pursue. For example, an agent that wants to achieve the goal of being in England must satisfy a number of sub-goals, such as buying a ticket, finding a hotel, getting a passport, etc. By virtue of deciding to go to England, the agent must consider all of these (and other) goals and must pick the best order in which to satisfy them.[6]

In general, all that this step requires is that the agent select a goal to attempt. Any reasoning method appropriate for the domain can be used to make the selection; Chapter 4 details a constraint-based formalism for ordering and selecting goals. Constraints are a useful and often natural way of modelling problems, and when local-search algorithms are used to solve these problems, they are naturally interruptible at almost any time, and can give useful feedback about the state of their processing. Other reasoning frameworks are possible, e.g. heuristic search, logical theorem-proving, uncertain/probabilistic reasoning, case-based reasoning, and numerous variations and combinations of such methods. The key point is that whatever reasoning technique is chosen, it must fit the resource constraints imposed by the domain and required by step three.

3.5.1 A Note on Terminology

Agents perform actions to achieve goals. For example, an agent might have the goal of being in England, and a sequence of actions such as "buy a plane ticket", "go to the

[6]Of course, a general-purpose agent such as a person will always have many different and unrelated goals to consider. For example, while planning a trip to England, I might realize that I am quite hungry and must decide if I should have lunch or phone the travel agent first. If I decide to have lunch later, it seems plausible that instead of always completely forgetting about having lunch (i.e. somehow deleting those goals from my memory such that my feeling of hunger must trigger them again for me to consider having lunch), the goal to have lunch is given a priority lower than the goals related to planning the trip. Thus, once the main goal for calling the travel-agent has been satisfied, the goal to have lunch will naturally appear as the next one, assuming no higher-priority goals have appeared in the meantime.

airport", "get on the plane", etc. will, if all goes well, cause the agent to end up in England. In a state-based model of planning, goals correspond to world-descriptions, and actions are transformations that change part of the description. This is a basic and important distinction in modern planning systems (e.g. see Russell and Norvig 1995).

Throughout this thesis we will only be considering simple atomic goals, so while the distinction between goals and actions still exists, it is not a vital one. We will assume a one-to-one mapping between goals and actions, and so we will talk about goals and actions as if they are the same thing. A successfully performed action will achieve the goal it is paired with, and we will sometimes refer to performing goals, which is understood as referring to the corresponding action.

3.6 Execution: When Should I Take a Turn?

Once a turn-taking agent has decided which goal to attempt, it must still take into account one extremely important factor: the other conversant! Since avoiding doubletalk and awkward pauses is important, an agent must be careful to take its turn at the right time in the conversation. To do this in human-human conversation, a person must be a good listener: she must look for the appropriate turn-yielding signals from the current speaker, signals such as pauses, changes in inflection, grammatical boundaries, gestures, etc. (Oreström 1983; Goldman-Eisler 1968). For example, a pause is often a good indicator of a speaker's willingness to give up the floor, but not always. Sometimes a pause simply indicates that the speaker is thinking, and so perhaps does not want to be interrupted. For human-computer conversation, we would ultimately like to develop a system that could take advantage of these same signals in order to engage in smooth conversational turn-taking.

Tension exists between the desire for an agent to execute its turn-taking goals, and

the desire not to haphazardly interrupt the other conversant. Sometimes, the agent may guess wrong (i.e. not interrupt when it should have interrupted), and, due to the time-bounded nature of turn-taking goals, be forced to drop a goal because its time of relevance has passed. The opposite problem is jumping in at the wrong time, which causes the conversation to flow less smoothly.

Chapter 5 discusses the concept of the conversational floor. A formal descriptive model is presented, which provides a clear and straightforward language for talking about turn-taking, and, implicitly, for related concepts such as initiative. It allows natural definitions of interruptions and disruptions, plus the turn-yielding and turn-ending signals are seen to be natural consequences of the underlying multi-agent, single-object model.

In practice, all agents are resource-bounded agents. Knowing when to speak requires knowing something about your state of processing. Most problem-solving procedures indicate when they are finished solving a problem, but it turns out that in many turn-taking systems an agent must have some idea about how near it is to solving a problem. This is necessary because another agent can request an agent to act before the agent has finished its processing, and, since we want to avoid awkward pauses, the agent should give some feedback about its processing state. Section 5.4 shows how an agent can use knowledge of its processing state to help tell it what to say.

3.7 Summary

This three-part model of turn-taking describes an architecture that a single agent can embody. In the following chapters, specific implementations will be presented that are geared towards resource-sensitive conversational systems. As discussed in Chapter 6, the higher-level distinctions that the three-part model makes can provide a useful way of looking at a broad range of interactive systems.

Chapter 4

What to Say: Ordering Multiple Actions

> You're out of order! This whole trial is out of order!
>
> *Arthur Kirkland, ... And Justice for All*

4.1 Introduction

This chapter covers the second step of the general turn-taking architecture introduced in Chapter 3. First, Section 4.2 will introduce the general action selection problem, and Section 4.3 presents presents a constraint-based framework for solving this problem. The constraint paradigm, including solution methods, are introduced, and then in Section 4.4 a precise statement of the next-action problem is given. Section 4.6 then presents a number of local search algorithms for solving the next-action problem, and gives the results of an empirical evaluation.

4.2 Choosing What to Do Next

Ever notice that *what the hell* is always the right decision? *Marilyn Monroe*

The traditional AI planning problem can be phrased in the following non-traditional way:[1] given a set of actions $A = \{a_1, \ldots, a_n\}$, an initial state I and a goal function G to be maximized, find a partially ordered subset A' of A that, when applied to I, results in a state that maximizes G.[2] Actions can be thought of as functions that transform states into other states, and so a partially ordered set of actions is applied to an initial state according to the order, with each successive action being applied to the state resulting from the previous action application. A *batch planner* (or just *planner*) is an algorithm that takes A, I, and G as input, and outputs a partially ordered subset of A called a *plan.* Planning is known to be a very hard problem in general (Bylander 1991), and heuristics or special domain knowledge are used whenever possible to achieve reasonable running times.

Now consider the related but different problem of finding just the *next* action to perform, which we refer to as the *next-action* planning problem (also called the *action selection* problem). In this case, we do not necessarily need to find an entire plan for optimizing G, just a next possible step in such a plan. This kind of planning can be useful in robotics, where a robot senses its local environment, and then makes a decision about what to do next; Maes (1990) shows that such planning need not be entirely reactive, but can involve deliberation about goals. A major advantage of such robots is that they operate more quickly than those based on general-purpose AI planners, and such robots can act reasonably (if simply) in an uncertain or changing environment.

[1] For a good introduction to traditional AI planning, see (Weld 1994).

[2] Often G is a particular state that the planner needs to find a path to. Treating G as a function to optimize slightly generalizes this perspective, and will be useful when we define the next action selection problem.

The input to a next-action planner is $N = <A, I, G>$, where $A = \{a_1, \ldots, a_n\}$ is a set of actions, I is the initial state of the world, and G is the goal function to be optimized. The output is the best action to perform next; sometimes, only one best action will be given, but other times more than one action is possible. By running a next-action planner multiple times in a row, we can simulate a batch planner; similarly, by running a batch planner and generating a complete plan but giving only the next requested step, we can simulate a next-action planner. Figure 4.1 gives an example of a next-action approach to solving a blocksworld problem.

A next-action planner can be useful in domains with a highly dynamic or uncertain world, such as in multi-agent systems (where agents can never be completely certain what other agents are thinking), or when plan execution is interleaved with plan generation. For example, in a dynamic world, the goal function G might change over time, e.g. if you form a plan to go to your favorite Italian restaurant, but discover half-way there that it is closed, you can change your goal and go elsewhere. A next-action planner handles such situations in a natural way by allowing different next-action planning problems at each time-step of interest. If $N_t = <A_t, I_t, G_t>$ is the next-action planning problem to be solved at time t, an agent operates by solving a sequence of problems $N_1, \ldots, N_t, \ldots$. Strictly speaking, there need be no relation between N_t and N_{t+1}, but in practice one expects adjacent N_t's will usually be similar, and so a solution to N_t can be used to help solve N_{t+1}.

4.3 A Framework for Action Selection in Discourse

Consider an agent that decides what to do next using a next-action planner. At time t, the agent must solve a next-action problem, i.e. select the best action from $A_t = \{a_1, \ldots, a_n\}$,

Penalty Matrix for G:
$\{on(\mathsf{A},\mathsf{B}), on(\mathsf{B},\mathsf{C}), on(\mathsf{C},\mathsf{table})\}$

$p(x,y)$		y = A	B	C	table
	A	·	0	0	1
x	B	1	·	0	1
	C	2	1	·	0

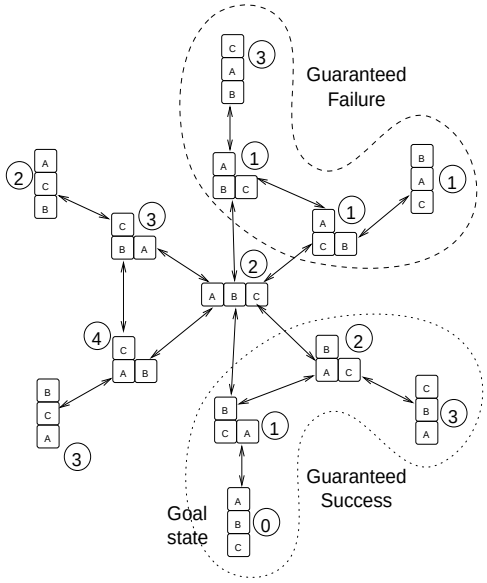


Figure 4.1: A blocks world state graph for 3 blocks. Two configurations are connected if one can be transformed to the other by moving one block. The blocks are assumed to rest on a table, and each configuration can be described using the $on(x,y)$, predicate, which means x is on top of (and touching) y. Any block can be on the table, but the table can never be on another block. The goal state is a single tower with A on top, B in the middle, and C on the bottom, and is described by $G = \{on(\mathsf{A},\mathsf{B}), on(\mathsf{B},\mathsf{C}), on(\mathsf{C},\mathsf{table})\}$. The given penalty matrix is defined for G, and a configuration is scored by adding the corresponding penalty values for each $on(x,y)$ predicate in its description. We use $p(x,y)$ to denote one entry in the penalty matrix, and P is the total penalty for a configuration. So, for example, $P(G) = p(\mathsf{A},\mathsf{B}) + p(\mathsf{B},\mathsf{C}) + p(\mathsf{C},\mathsf{table}) = 0$. This graph can be searched according to the follow greedy method: pick a random starting state, and then always move to an adjacent state that has the lowest penalty score (break ties randomly). If this algorithm ever strays into one of the "guaranteed failure" states, it will become trapped forever and never escape to find the goal; if it ever reaches one of the "guaranteed success" states, the goal will be found. If the algorithm ever gets to the middle state (i.e. where all the blocks are on the table), there is a two in three chance it will get stuck in the forbidden area. Clearly, great care must be taken in action selection planning systems to avoid such traps.

the actions that can possibly be attempted at time t.[3] An agent could use a batch planning method to solve the problem, but that would likely be inappropriate in an uncertain or time-pressured domain because much re-planning could be required when the agent obtains more knowledge about the world. Instead, we focus on methods that quickly select actions at the cost of some solution quality. Not every domain of application can tolerate such a trade-off, but many interesting ones can; in particular, discourse can be thought of as a multi-agent planning (and plan recognition) problem that can tolerate errors in the sense that if one agent makes a mistake, there is often the chance for another agent to catch it, or at least point out that a problem might exist. Our goal is not just to solve the action selection problem, but to solve it using methods that are sensitive to time pressure and a dynamic/uncertain environment, so the solution strategies of interest to us must address these concerns.

4.3.1 Ordering Actions with Constraints

Selecting the next action is, in general, a difficult reasoning problem, and one must choose some general framework to work within. We have chosen the paradigm of constraint satisfaction (Tsang 1993), for a number of reasons. Constraint-based reasoning is remarkably flexible; many important combinatorial problems can be treated as sub-classes of finite constraint satisfaction problems (CSPs): propositional satisfiability, graph coloring, configuration, and various scheduling problems. We first define the basic constraint satisfaction problem, and then generalize these to constraint optimization problems.

[3] Note that just because an action can be attempted does not mean that it will necessarily be executed successfully, or with a known effect. In the most general case, for an action to be *possible* means it is possible for it to be attempted.

Constraint Satisfaction Problems

To solve a *constraint satisfaction problem* (*CSP*) P, one must find a set of values for variables $V_1, \ldots, V_n$, chosen from corresponding domains of possible values $D_1, \ldots, D_n$, such that none of the values assigned to $V_1, \ldots, V_n$ break any constraints. Any set of variables may have a constraint imposed on them, and constraints are usually specified by listing the allowable (or unallowable) values these variables can be simultaneously assigned. In other words, a constraint on a set of variables is a set of tuples that explicitly list their possible values. We only deal with *finite* CSPs, which means the number of variables and constraints on them is finite, plus all domains are finite; also, when constraints are treated as sets of tuples, these sets must be finite. A common simplifying assumption is to deal only with *binary* constraints that restrict values of pairs of variables; binary CSPs can be drawn as graphs, with nodes representing variables, and edges representing constraints. Any CSP with n-ary constraints can be transformed into one with just binary constraints.

Definition 1 *A finite constraint satisfaction problem (CSP) consists of:*

1. *A finite set of variables, $V_1, \ldots, V_n$, $n > 0$; a variable can be assigned at most one value;*

2. *A corresponding set of domains, $D_1, \ldots, D_n$, such that the domain of V_i is D_i and V_i can only be assigned values from D_i;*

3. *For all non-empty subsets V of $\{V_1, \ldots, V_n\}$ containing k variables, C_V is the set of k-tuples that specify all of the assignments that the variables in V can simultaneously take on. Each value of a tuple must occur in its corresponding domain (i.e. if the jth entry of a tuple of C_V corresponds to a possible value for V_i, then that entry must be in D_i). V obeys, or satisfies, constraint C_V if the values assigned to the variables in V are a tuple in C_V; an unsatisfied constraint is said to be broken.*

A complete assignment of values to variables is denoted $\{V_1 = d_1, \ldots, V_n = d_n\}$, $d_i \in D_i$; *a partial assignment is an assignment where some* V_i*'s are unassigned. A solution to a CSP is a complete assignment* $A = \{V_1 = d_1, \ldots, V_n = d_n\}$ *that breaks no constraints.*

In practice, if a set of variables V can take on any possible value (i.e. C_V is the cartesian product of the domains of the variables), no explicit constraint is given. Thus, while the above definition requires one set of constraints for every possible proper subset of $\{V_1, \ldots, V_n\}$, usually most of these constraints allow for any possible set of values, or are implied by other constraints. In a binary CSP with n variables, at most n unary and $n(n-1)/2$ binary constraints can be specified.

The above formulation of CSPs has proven to be a computationally convenient model for many constraint problems, and a number of efficient simplification techniques, such as arc-consistency,[4] have been developed that transform CSPs into simpler CSPs with the same set of solutions. Various backtracking methods can be used to find one or all solutions to a CSP, and backtracking interleaved with some form of arc-consistency is often a good general-purpose and complete way of finding all solutions to a CSP. The typical method for solving a CSP in this way is to iteratively select variables and assign them values, and after each variable assignment remove values from the domains of unassigned variables that cannot possibly be part of a complete assignment. If any unassigned variable's domain is empty, then one or more of the assigned variables must be unassigned and given a new value. The behaviour of such algorithms can be improved through good heuristics for choosing variables and domain values, along with efficient

[4]The arc corresponding to (V_i, V_j) is *arc-consistent* if for every value x in the domain of V_i, there is a corresponding value y in the domain of V_j such that $V_i = x, V_j = y$ is an allowable assignment (i.e. in the binary constraint between V_i and V_j). A CSP is arc-consistent if all of its individual arcs are arc-consistent. Although it will not be used in this thesis, the idea of arc-consistency is important because it is sometimes more efficient than pure backtracking methods and, in some special cases, can completely solve a CSP. Making a CSP arc-consistent is often a first pre-processing step in backtracking algorithms, since even if it does not completely solve it, it can significantly reduce the search space. Kumar (1992) and Tsang (1993) discuss this in greater detail.

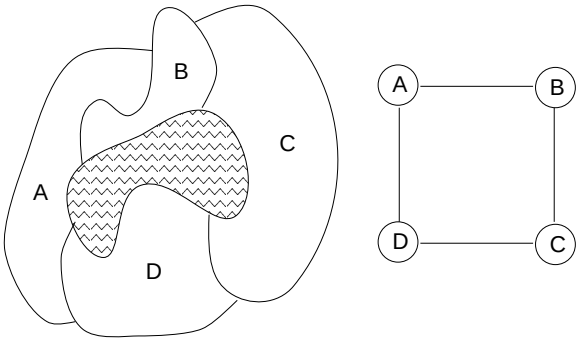


Figure 4.2: Four countries in a cycle. That's water in the middle.

data structures for keeping track of what variables and values have been tried. Tsang (1993) is a good reference for all these topics, and more.

Examples

As a concrete example, consider the problem of coloring n countries on a map with 3 colors such that no two adjacent countries have the same color. This is not always possible to do perfectly, but the number of clashes can be minimized. For country i, V_i holds its assigned color. Each variable must take on one of three possible colours, so all the domains are equal, i.e. $D_1 = \ldots = D_n = \{0, 1, 2\}$. We specify the constraints as follows, showing the allowable values that the given pair of variables may take on:

$$C_{V_i,V_j} = \begin{cases} \{(0,1),(1,0),(0,2),(2,0),(1,2),(2,1)\} & \text{if } i,j \text{ adjacent} \\ \{(0,0),(1,1),(2,2),(0,1),(1,0),(0,2),(2,0),(1,2),(2,1)\} & \text{otherwise} \end{cases}$$

If, on our map, we have four countries A, B, C, D connected in a cycle as in Figure 4.2 (i.e. A is adjacent to B and D, and C is adjacent to B and D), a number of assignments satisfy all constraints, e.g. $\{V_A = 0, V_B = 1, V_C = 2, V_D = 1\}$, or $\{V_A = 0, V_B = 2, V_C = 0, V_D = 2\}$. Figure 4.3 shows a simple PROLOG program that solves this problem.

```
%
% domain values
%
domV1(red). domV1(green). domV1(blue).  % V1 domain: red, green, blue
domV2(red). domV2(green). domV2(blue).  % V2 domain: red, green, blue
domV3(red). domV3(green). domV3(blue).  % V3 domain: red, green, blue
domV4(red). domV4(green). domV4(blue).  % V4 domain: red, green, blue

%
% constraints on variables
%
constraints(V1,V2,V3,V4) :-
        domV1(V1), domV2(V2), domV3(V3), domV4(V4),
        V1 \== V2, V2 \== V3, V3 \== V4, V4 \== V1.
```

Figure 4.3: A PROLOG program for solving the graph three-colouring problem in Figure 4.2. By calling `constraints(V1,V2,V3,V4)`, the program finds values for `V1`, `V2`, `V3`, and `V4` that meet the colouring constraints; in total, there are 18 different solutions to this problem (out of $3^4 = 81$ possible colourings). The method amounts to just generate and test, and so is infeasible for larger problems. Also, this program will fail if it does not find a perfect three-colouring, so it cannot find a colouring with a minimum number of constraint violations.

As another example of the use of CSPs, consider solving the following puzzle:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Each letter is a digit from 0 to 9, different letters are different digits, and a number cannot start with a 0. This can be formulated as a CSP by treating each of the 6 different letters, D, E, N, O, R, Y, as CSP variables each with domain $\{0, \ldots, 9\}$, and the letters S and M as CSP variables with domain $\{1, \ldots, 9\}$. The PROLOG program shown in Figure 4.4 solves this problem by doing addition in the ordinary right to left fashion, keeping track of which variables it has assigned. The key idea is to select values for digits non-deterministically with the `choose` function, which both chooses a value from a given list and remembers what values remain unchosen in case the program later "fails" and backtracks to the choose statement, where the next unchosen value is selected. PROLOG supports general-purpose non-determinism, so this is easy to implement.

Constraint Optimization

A *constraint optimization problem* (COP) is a CSP where not all constraints need be satisfied. We associate with each constraint a penalty for breaking it, and define a *penalty function* P that, for any partial or complete assignment of values to variables, returns the sum of the penalties of the disobeyed constraints. Thus, to solve a COP, one tries to minimize P over a complete assignment; a perfect solution to a COP is a complete assignment of values to variables S such that $P(S) = 0$. Constraints that can never be broken are called *hard* constraints, and constraints that can be broken are called *soft* constraints (or preferences).[5]

[5] We note that in PROLOG, for example, it is quite simple to write programs that solve hard constraint problems, but it offers no special advantages for handling problems with soft constraints.

```
digits([0,1,2,3,4,5,6,7,8,9]).
choose(L,H,LminusH) :-          % non-deterministically selects a
        member(H,L),            % member of L, putting that selection
        delete(L,H,LminusH).    % into H, and returns in variable
                                % LminusH the contents of L but with H
                                % removed

splitdigits(Num,D1,D2) :-       % Num is an integer from 0, ..., 99
        Num < 100,              % (i.e. a 1 or 2 digit number). In D1 is
        0 =< Num,               % returned the tens digit of Num (0 if Num
        D2 is Num mod 10,       % is only a single digit), and in D2 is
        D1 is Num // 10.        % returned the ones digit of Num.

constraint(S,E,N,D,M,O,R,E,M,O,N,E,Y) :-
        digits(D0),
        choose(D0,D,D1),
        choose(D1,E,D2),
        DplusE is (D+E),
        splitdigits(DplusE,CarryDE,Y),
        choose(D2,Y,D3),
        choose(D3,R,D4),
        choose(D4,N,D5),
        RplusNplusCarryDE is (R+N+CarryDE),
        splitdigits(RplusNplusCarryDE,CarryRN,RN),
        E == RN,
        choose(D5,O,D6),
        EplusOplusCarryRN is (E+O+CarryRN),
        splitdigits(EplusOplusCarryRN,CarryEO,EO),
        N == EO,
        choose(D6,S,D7), S \== 0,
        choose(D7,M,_), M \== 0,            % _ is an anonymous variable
        MplusSplusCarryEO is (M+S+CarryEO),
        splitdigits(MplusSplusCarryEO,CarryMS,MS),
        O == MS,
        M == CarryMS.
```

Figure 4.4: A PROLOG program for solving the SEND MORE MONEY puzzle. This program can be thought of as describing a correct solution to the problem, and PROLOG automatically loops through variable assignments, stopping only when it finds a full set of variable assignments that meets all the constraints. In contrast to Figure 4.3, a different technique for storing domains values is used, and assignments and constraint checks are interleaved. This interleaving allows partial assignments to be evaluated and immediately rejected if they can be recognized as not being part of a solution. The one solution found is: $S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2$.

A binary CSP is a CSP that has only binary or unary constraints. For simplicity, and because the next-action problem defined in Section 4.4 requires nothing more, we only consider binary COPs.

Definition 2 *A finite binary constraint optimization problem (COP) is a binary CSP with the following additions:*

1. *For each different pair of variables V_i, V_j, a basic penalty function p_{V_i,V_j} is defined such that $p_{V_i,V_j}(d_i, d_j) \geq 0$ for all $d_i \in D_i$ and $d_j \in D_j$;*

2. *A penalty function P that calculates the penalty for a partial (or complete) assignment $A = \{V_1 = d_1, \ldots, V_k = d_k\}$:*

$$P(A) = \sum_{d_i, d_j \in A} p_{V_i,V_j}(d_i, d_j)$$

A solution to a COP is a complete assignment A that minimizes $P(A)$.

CSPs can be thought of as a special case of a COP, where all the constraints are hard and a solution S is a complete assignment that violates no constraints, i.e. $P(S) = 0$. Solving COPs requires different techniques than solving regular CSPs (Tsang 1993). However, local search methods can be used to solve both CSPs and COPs, without any major change in the basic algorithm (Wallace and Freuder 1995). In this thesis, we do not use this general COP framework, but instead deal with a more specific optimization problem as discussed in Section 4.4.

Dynamic CSPs

If we let P_i represent a particular CSP, then we can define a *dynamic CSP* to be a series of CSPs $P_1, P_2, \ldots, P_n, \ldots$. Intuitively, one can imagine an agent operating in a

changing world, where each change it perceives causes it to create a new CSP for it to solve. Strictly, there need be no relation between CSPs P_i and P_{i+1} in a dynamic CSP, but in most domains adjacent CSPs will tend to be similarly structured. This implies that when solving P_{i+1}, the agent can try to exploit its solution to P_i so it does not need to solve P_{i+1} completely from scratch.

Even slightly changing the structure of a CSP can significantly change the possible solutions. For example, if we change one letter in the "SEND+MORE=MONEY" puzzle (Figure 4.4) so it becomes "SEND+MORE=HONEY", the modified version has no solutions. Intuitively, *tightening* constraints between variables will result in fewer solutions, while *loosening* constraints will result in more.[6]

For general CSPs, a number of basic modifications can be made to them that preserve a given solution. We summarize these in a theorem.

Definition 3 (Constraint tightness) *For any constraint C_V of a finite binary CSP P, C_V is* loosened *by adding new tuples, and C_V is* tightened *by removing tuples.*

Theorem 1 (Solution-preserving modifications) *Let S be a solution of CSP P. The following modifications to P all preserve S as a solution:*

- *Adding a new value to any domain D_i;*
- *Removing a value from D_i that is* not *the value assigned to V_i by S;*
- *Loosening any constraint;*
- *Tightening any constraint C_V by removing any tuple from C_V except tuples assigned to the variables of V by S.*

[6] This particular line of reasoning has been used to empirically exhibit phase transitions in randomly generated CSPs (APES 1998).

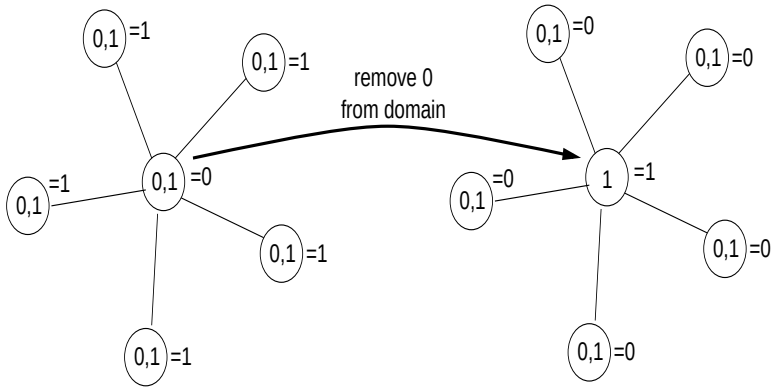


Figure 4.5: Dynamic CSP example graph. Edges represent not-equal constraints. Removing 0 from the domain of the center node forces it to be assigned 1. That then requires that each adjacent variable change its assignment from 0 to 1.

Removing a variable V_i from P is the same as totally loosening all the constraints C_V where V_i is in V. If P_0 is the initial problem of a dynamic CSP P and has a solution S_0, and every P_{i+1} is constructed from P_i by one or more of the solution-preserving modifications of Theorem 1, then S_0 is a solution for each P_i.

The difficult cases for solving a dynamic CSP involve changing parts of P that affect values assigned by a solution $S = \{V_1 = d_1, \ldots, V_n = d_n\}$. If, say, d_1 is removed from D_1 to create P', then S cannot be a solution of P'. Whether or not P' has any solutions, similar to S or otherwise, depends on the structure of P. Figure 4.5 shows a simple CSP where removing an assigned value from a domain results in a CSP whose solution has no values in common with the original problem solution.

4.3.2 Local Search

Local search is a general-purpose optimization strategy that repeatedly modifies a fully instantiated set of CSP variables in hopes of finding a better solution. Such methods often combine the characteristics of hill-climbing and greedy algorithms, although numerous

```
procedure local-search()
  S ← an initial complete assignment
  loop
    if S is a solution then
      done
    elsif S not a solution then
      S ← neighbor of S
    end if
  end loop
end local-search
```

Figure 4.6: The generic local search algorithm. A well-chosen initial solution can greatly help local search, so it often pays to choose this value carefully. The loop typically continues until a solution is found, or a bail-out condition is reached (e.g. stop after 1000 iterations). Local search often gets stuck in cyclic behavior, or converges to a local instead of a global optimum (see example in Figure 4.1). Typical tricks for combating these problems are to add some randomness to selecting S, or to repeatedly run local search on different starting values.

variations are possible (e.g. simulated annealing). Figure 4.6 gives the general algorithm.

The Min-conflicts Heuristic

Minton et al. (1992) present a remarkably simple local search heuristic that can be used to solve general CSPs and COPs. It has performed extremely well in certain scheduling problems, and can efficiently find solutions to the n-queens problem for n over a million in a few seconds.[7]

We first introduce some terminology. The CSP assignment $V_i = d_i$ is said to be *conflicted* if it participates in a constraint violation, i.e. if there is some other assignment $V_j = d_j$ such that $(d_i, d_j) \notin C_{V_i,V_j}$. Given an assignment $A = \{V_1 = d_1, \ldots, V_k = d_k\}$, the *conflict count* is the number of conflicted variables in A. We call changing $V_i = d_i$

[7]The n-queens problem is known to be in complexity class P, so it is not considered a hard problem. However, it is a common test problem for CSP algorithms, and most general methods have difficulty solving it for a couple of hundred queens.

to $V_i = d'_i$ in an assignment a *repair*. One repair is better than another if it decreases the conflict count more, and the best repair for a particular variable is one that increases the conflict count the least (i.e. decreases it the most). Min-conflicts is a local search algorithm that works on complete assignments and at each iteration it looks at all the best repairs for each individual variable, and actually performs the best of the best repairs; ties are broken randomly.

The success of this method depends very much on the problem domain. Somewhat surprisingly, it works very well for the n-queens problem,[8] and a variation of this method is perhaps the most efficient algorithm for solving satisfiable boolean satisfiability problems (Selman et al. 1992; Selman et al. 1994; Kautz and Selman 1996). While it is straightforward to apply to general CSPs and COPs, typically much tuning is required to coax out the best performance, although progress has been made on applying local search to general planning problems (Ambite and Knoblock 1997; Bonet et al. 1997)

Local search applies naturally to both dynamic CSPs and COPs. Suppose local search has currently found S_i to be the best solution to COP P_i. If the process continues, a better solution might be found, but before that the problem changes to P_{i+1}. We can apply local search to P_{i+1} by using S_i as the starting solution.

[8] In support of solution techniques that throw out part of a solution space, Pearl (1984, p.17–18) writes the following:

> Imagine, for example, that someone decides to solve [the 8-queens] puzzle by placing eight queens in some random initial position and then perturb the original position iteratively by moving one queen at a time. ...as in the case of trying to find an address in a telephone directory without a name, we can reject individual objects but not large chunks of objects.

We note the irony that Pearl's argument, while plausible in general, fails spectacularly if one "perturbs" the chess board according to the min-conflicts procedure (or slight variations; see Gu 1992). Successful hill-climbing heuristics are often "dumb and fast", typically using randomness so they do not suffer from "biases" and to escape from local extrema that plague all search methods. The telephone directory analogy assumes an "intelligent and slow" search agent, which is plausible in general, but should not be taken to exclude the possibility that dumber, faster, and in many ways better local search methods can be used.

4.4 The Next-action Problem

To avoid having to solve general COPs, we narrow the description of the problem such that it still does what we need it to do, but takes on a form similar to the traveling-salesman problem and so allows more specific solution methods to be exploited. In the language of CSPs, the next-action problem consists of k CSP variables, $V_1, \ldots, V_k$, each with the same domain of actions $D = \{d_1, \ldots, d_n\}$. The next-action problem uses only binary constraints which are encoded in a set of basic penalty functions G_{V_i,V_j}; however, for simplicity, we usually only talk about a single basic penalty function G in which the penalty scores for all possible pairs of values is encoded. As in the definition of P for COPs (Section 4.3.1), by summing over all pairs of assigned values, a score for any partial or complete assignment can be calculated.

The local search methods we consider for solving the next-action problem are inspired by the very successful k-opt exchange heuristics for the traveling salesman problem (TSP for short) (Lin and Kernighan 1973; Reinelt 1994). Recall that the TSP asks one to find a shortest complete tour of a group of cities, where the distance between each city is known. Typically, the distance between cities is given as an $n \times n$ matrix of non-negative numbers, and the problem is to find a permutation of the cities that minimizes the total sum of the distances between adjacent cities (and the first and last cities). Figure 4.7 gives a precise description of the problem.

The TSP can be thought of as a COP by treating the distances between cities as penalties. The total penalty for any list of cities is the sum of all the penalties between adjacent pairs of cities, i.e. if the ordering is $<c_4, c_3, c_1, c_2>$, the total penalty is

$$\text{TSP total penalty} = p(c_4, c_3) + p(c_3, c_1) + p(c_1, c_2) + p(c_2, c_4)$$

The next-action problem is very similar to the TSP, the essential difference being that

TRAVELING SALESMAN
INSTANCE: A finite set $C = \{c_1, c_2, \ldots, c_m\}$ of "cities", a "distance" $d(c_i, c_j) \in Z^+$ for each pair of cities $c_i, c_j \in C$, and a bound $B \in Z^+$ (where Z^+ denotes the positive integers).
QUESTION: Is there a "tour" of all the cities of C having total length no more than B, that is, an ordering $<c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)}>$ of C such that

$$\left(\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right) + d(c_{\pi(m)}, c_{\pi(1)}) \leq B?$$

Figure 4.7: Specification of the traveling salesman problem (Garey and Johnson 1979).

the total penalty is calculated in a different way. In the TSP, the total penalty of an ordering of cities is the sum of the penalties of *adjacent* cities, while in the next-action problem the total penalty is the sum of the penalties of *all pairs* of cities:

$$\begin{aligned} \text{Next-action total penalty} \quad = \quad & p(c_4, c_3) + p(c_3, c_1) + p(c_1, c_2) + \\ & p(c_4, c_1) + p(c_3, c_2) + \\ & p(c_4, c_2) \end{aligned}$$

The TSP-tour requires a connection between the starting and ending cities of the tour, but the next-action problem only requires a *path* from the first goal to the last goal; thus the first line of the next-action total penalty sum is almost the same as the TSP total penalty calculation.

The TSP penalty function is inadequate for ordering actions because it only assigns a penalty for out-of-order actions that are adjacent to each other. For example, we would want to assign a high penalty for eating an apple before possessing it. The TSP penalty function would only assign a penalty if action *eat-apple* was followed immediately by *get-apple*. However, no penalty would be assigned to the action sequence *eat-apple*, *twiddle-thumbs*, *get-apple*. This is clearly unacceptable, so the penalty function for the

Given: n actions, $A = \{a_1, \ldots, a_n\}$
k variables to fill, $1 < k \leq n$
a penalty function $p(a_x, a_y) \geq 0$ that gives the penalty for a_x coming before a_y
Find: a $1-1$ function $\sigma : \{1, \ldots, k\} \rightarrow \{1, \ldots, n\}$ that minimizes

$$G(A) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} p(a_{\sigma(i)}, a_{\sigma(j)})$$

Figure 4.8: The next-action problem.

next-action problem is designed to assign penalties for any pair of actions that are out of order, no matter how far apart they are.

This close connection between the TSP and next-action problems suggests that solution methods for the TSP may be appropriate for solving the next-action problem. In particular, local search algorithms for the TSP problem can be applied in a fairly direct way to the next-action problem. Such algorithms look at pairs of actions and then swap them according to some heuristic criteria. Action swaps form the atomic algorithmic steps, between which the variables are always fully instantiated, so the algorithms can be interrupted at almost any instant, making them natural *anytime* procedures. Also, if new actions are added or removed, or the goal function G changes, the algorithm can continue naturally, without necessarily needing to abruptly change its behaviour.

Figure 4.8 defines the next-action problem as a constraint optimization problem similar to the path asymmetric traveling salesman problem.[9] This formalism allows us to partially order actions, where weights specify the strength of an ordering. If action a must be executed before b — represented by $a < b$ — then we assign a high penalty to

[9]The asymmetric traveling salesman problem is the traveling salesman problem where the distance between cities A and B need not be the same as the distance between cities B and A. The path traveling salesman problem is the traveling salesman problem where the distance between the starting city and ending city are not counted in the scoring function. The path asymmetric traveling salesman problem is just the combination of these two problems.

$b > a$. We cannot directly specify how soon b ought to be executed after a because the penalty function p depends only on a and b, not on i or j. However, if another action c is constrained to come between a and b, then, assuming no other constraints, the best ordering is $<a, c, b>$, which effectively requires b to *not* appear directly after a. Instead, if all that is specified is that a come before b and c, the orderings $<a, b, c>$ and $<a, c, b>$ score the same. We cannot directly specify, say, that a be closer to c than to b.

Inconsistent orderings can be specified. For example, if we require $a < b$, $b < c$, and $c < a$, then, assuming 1 penalty point per broken constraint, half the orderings have a penalty of 1, while the other half have a penalty of 2. No perfect ordering exists, but a set of least penalized orderings can be found. Representing such inconsistencies is important because in domains where the world is changing or uncertain, it is quite possible for an agent to have a strictly inconsistent view of the world, and yet still be expected to act rationally. Instead of searching for a possibly non-existent perfect solution, an agent should instead try to find the *best possible* solution, namely the action ordering that minimizes the total penalty.

4.4.1 Terminology and an Example

Along with the CSP terminology just introduced, we will describe the next-action problem from a goal-oriented point of view. A *goal* in this context is just an object with a *goal-type*, and the penalty function p actually only cares about the type of the goal; so, for example, if goals G_a and G_b are both of the same type, then $p(G_a, G_b) = 0$.[10] We distinguish between the *goal pool*, where all n goals are kept with no concern for order, and *working memory*, represented by the k ordered variables. We will sometimes refer to individual locations in working memory as *slots*, or *variables*, to be filled, and designate

[10] In other applications, other goal attributes might be relevant, such as the time of arrival of the goal, the amount of time it has been in memory, its content, etc.

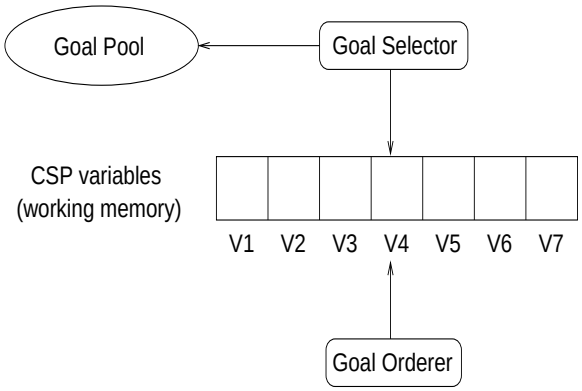


Figure 4.9: Pictorial representation of the turn-taking problem.

slot i by CSP variable V_i. Figure 4.9 shows a pictorial representation of the problem.

Consider an example. Suppose there are $n = 10$ actions and $k = 5$ working memory slots, plus 10 different goal types. The 10 goals in the goal pool are $g_1, \ldots, g_{10}$, and each goal is of a different type, i.e. g_i is of type i. Working memory consists of 5 slots, $V_1, \ldots, V_5$. We define the penalty function $p(g_i, g_j) = 1$ if i is composite and j is prime, otherwise $p(g_i, g_j) = 0$; Figure 4.10 shows this function in matrix form. Here are some example orderings and their associated penalty scores:

$$
\begin{aligned}
< g_1, g_2, g_3, g_4, g_5 > &= (p(g_1, g_2) + p(g_1, g_3) + p(g_1, g_4) + p(g_1, g_5)) + \\
&\quad (p(g_2, g_3) + p(g_2, g_4) + p(g_2, g_5)) + \\
&\quad (p(g_3, g_4) + p(g_3, g_5)) + \\
&\quad (p(g_4, g_5)) \\
&= (1 + 1 + 1) + (0) + (0) + (1) \\
&= 4 \\
< g_5, g_4, g_3, g_2, g_1 > &= (0) + (1 + 1) + (0) + (0) \\
&= 2
\end{aligned}
$$

$i \setminus j$	1	2	3	4	5	6	7	8	9	10
1	0	1	1	0	1	0	1	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	1	1	0	1	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	1	1	0	1	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	1	1	0	1	0	1	0	0	0
9	0	1	1	0	1	0	1	0	0	0
10	0	1	1	0	1	0	1	0	0	0

$$p(g_i, g_j) = \begin{cases} 1 & \text{if } i \text{ is composite and } j \text{ is prime} \\ 0 & \text{otherwise} \end{cases}$$

Figure 4.10: Matrix form of the example penalty function.

$$\begin{aligned} < g_4, g_8, g_5, g_6, g_1 > &= (1) + (1) + (0) + (1) \\ &= 3 \\ < g_2, g_4, g_6, g_8, g_{10} > &= (0) + (0) + (0) + (0) \\ &= 0 \\ < g_9, g_7, g_5, g_3, g_1 > &= (0) + (0) + (0) + (0) \\ &= 0 \end{aligned}$$

This shows that more than one optimal ordering is possible; note that an optimal ordering need not be 0 (e.g. every value of p might be greater than 0).

One difficulty in using the next-action problem as a planner is that the knowledge must all be encoded in the penalty matrix. In Appendix A we show a useful way to use a computer to assist in generating a penalty matrix, so that it need not be done entirely by hand. Experience with SG, the course-advising system discussed in Chapter 6, shows that the initial matrix generated by such a method might not be perfect, and so some

hand-tuning of the final matrix might still be required. However, automating at least part of the process will likely make this a less time-consuming task.

4.5 Solving the Next-action Problem

Recall that we do not want to simply find an algorithm to quickly solve large instances of the next-action problem, but prefer algorithms that are sensitive to the resource bounds imposed by the domain of application. Using the discourse domain (where an agent must choose what action to pursue on its next turn) as our guide, we want an algorithm that:

- Can be interrupted at almost any time, and be able to return a reasonable solution whose quality is proportional to the amount of time the algorithm ran;
- Can take advantage of any excess time it might have. For example, in a conversation one might be ready to speak, but must wait for the other conversant to finish their speaking turn. Ideally, that time should be spent speculating about what future actions could be performed by ordering the unselected actions as best as can currently be done;
- Makes some promise about how it will behave. In particular, we would prefer an algorithm that does not change its next action at arbitrary times during its processing. Instead, we would like to be able to know that, at some point, the algorithm has chosen the next action, and will not change that action unless important new information arises. This is related to the previous point in that such promises would allow us to distinguish between when the agent is ready to act, and when the agent needs more time to select its next action.

The algorithms discussed in Section 4.6 were chosen with these considerations in mind.

Another interesting point about the domain of application is the fact that people are not always ready to speak when they are expected to speak. For example, if someone asks you for the answer to the question *What is 25 times 25?*, you will likely have to pause to think for a moment.[11] This is not infrequent in ordinary conversation, and people sometimes use filler-phrases like *ummm*, *ahhh*, *let me see...*, to avoid an awkward pause. This suggests that whatever algorithm it is that people use to decide what to say next in a conversation, it might not be a purely anytime one, since there is often some minimal level of quality that people want to reach before saying something. Thus, in the next section, we will consider algorithms that cannot be interrupted while they are filling the *first* working-memory slot, but can be interrupted at any time after that.

4.5.1 Deleting Actions

The next-action problem is only concerned with finding the best ordering of actions. When the agent actually selects an action and executes it, that action, and possibly others, must be deleted from working memory. In general, when an action is accomplished, it must be deleted from working memory, and so too must any actions which it *dominates*.[12] We assume that domination relations are either set ahead of time, or can be calculated quickly by rules or templates when new actions appear in working memory. The *SG* system described in Section 6.4 shows how this technique fits into the operation of a turn-taking agent that works by solving the next-action problem.

4.5.2 Reasoning in a Dynamic Environment

Local search works naturally in a dynamically changing domain. If new actions are added to a next-action problem at arbitrary times, then they can simply be added to working

[11] 625.

[12] Action α dominates action β if successfully completing α means it is no longer necessary to do β.

memory where they will then be automatically placed by the next pass of the local search algorithm. Consider a sample course-advising dialog, where a student makes these two utterances:

Dialog 4.1

Student: Is CS100 offered this term? I need to fulfill my math credits.

It happens that CS100 cannot be used to fulfill any math requirements, since it is a low-level computer course. If the student only asked *Is CS100 offered this term?*, then the system might respond *Yes. The times it is offered are....* But the second utterance raises a more important problem that should be dealt with first, so a better answer to the above example would be *CS100 cannot be used to satisfy math requirements....*

An advising system that uses the next-action problem would operate as follows. The first student question would trigger various goals to appear in working memory, including at least one that will generate an answer to the question. As soon as a new goal is added to working memory, the local search algorithm can go to work. After the student has issued the first utterance, working memory will settle into a state where, if the system were required to act, it would respond with *Yes. The times it is offered are....* When the user makes the second utterance, new goals are added to working memory, and it will be reordered. In particular, the action to answer the first question is moved out of the initial position and replaced by one corresponding to clearing up the misconception displayed by the second utterance. Depending on the algorithm and the number of goals it is ordering, the system might not completely finish ordering working memory by the time the user begins their second utterance. The algorithm will proceed the same as before, except it will not (in the same way, at least) ever settle into asking the user the first question.

4.6 Algorithms for Solving the Action Selection Problem

Given the similarity of the traveling salesman problem and the next-action problem, the numerous and varied solution methods developed for the traveling salesman problem (Reinelt 1994) could be applied to the next-action problem. A study to determine the best local search algorithm would be beyond the scope of this thesis, since here it forms just one step of the larger turn-taking model. However, in order to get a feel for the behaviour of different basic kinds of local search algorithms, this section presents the results of a small empirical study done on a set of local search algorithms. The algorithms, listed in the following section, were chosen mainly due to their simplicity and similarity to effective algorithms for the traveling salesman problem. The simpler algorithms were chosen with a desire to see if a relatively weak but efficient algorithm would result in "good-enough" solutions of the next-action problem. The results suggest that most of the simple algorithms are too simple, and so this fact was used in the SG system (Section 6.4) which uses only the best-performing algorithm from this study. Another interesting result was that the standard two-opt algorithm, which works well for the traveling salesman problem, performed comparatively poorly on the next-action problem.

The algorithms listed in the following sections were tested empirically on the action selection problem of Figure 4.8. The number of actions is n, the number of variables is k, and t different action types. In each experiment, $n = k = 15$, and $t = 5, 10, 15, \ldots, 50$ action types were used. These values were chosen mainly because they represent a reasonably large problem size for the SG system of Section 6.4, and n and k were kept equal because they do not vary in SG, and a reasonable and systematic way for varying so many parameters is not obvious. For each problem, a $t \times t$ penalty matrix was generated with each non-diagonal entry randomly set to 1 or 2 (with 50% probability), and the diagonal entries were all set to 0. Each problem instance has its own random starting collection

```
procedure GenericSolve(I,H)
  while I.hasMoreElements() do
    (V_i,V_j)←I.nextElement()
    if H(V_i,V_j) is true then
      swap V_i and V_j
    end if
  end while
end GenericSolve
```

```
procedure IteratedGenericSolve(I,H)
  while penalty decreases do
    GenericSolve(I,H)
  end while
end IteratedGenericSolve
```

Figure 4.11: The generic next-action solving template. I is an iterator that returns pairs of variables, and H is a heuristic function that returns true when it decides to swap two given variables, and false otherwise.

of goal (types). Results are given in Section 4.6.9.

Most of the algorithms listed below are straightforward specializations of the abstract template shown in Figure 4.11.[13] H is a heuristic function that decides when a given pair of variables should be swapped. I is an *iterator*,[14] which defines the order in which pairs of variables are examined. In general, an iterator can be thought of as a collection of items, and every time $I.nextElement()$ is called, one of the items is removed from I and returned. $I.hasMoreElements()$ is false exactly when I has at least one element remaining.[15]

There are many ways to define iterators. For example, an iterator that goes through the numbers $1, \ldots, n$ can be defined explicitly as the sequence $I_n = <1, 2, \ldots, i, \ldots, n>$. I_n iterates through the numbers $1, \ldots, n$ in ascending order, so the ith call to $I_n.nextElement()$ returns i. Also, $I_n.hasMoreElements()$ is true if $I_n.nextElement()$ has been called $n-1$ or fewer times, and false otherwise.

Iterators can be designed to emit elements in any order, even repeating values. We

[13] The Greedy algorithm and Two Opt are not. The Greedy algorithm can be written in this form, but the way it is written below is clearer. For Two opt, see Section 4.6.4.

[14] The concept of an iterator is common in object-oriented design (Booch 1994), and captures many iterative scenarios; for example, the user in a computer dialog system can be thought of as an iterator containing a collection of utterances.

[15] The notation used here follows the Java `java.util.Enumeration` interface.

are mainly interested in iterators that output pairs of integers because these pairs are used to indicate which elements to swap. A very common order is to generate each pair exactly once in left-to-right order, as printed out by the following loop:

```
for i ← 1 to k-1 do
  for j ← i+1 to k do
    print (i, j)
  end for
end for
```

The pairs printed are: $(1,2),(1,3),\ldots,(1,k),(2,3),\ldots,(k-1,k)$. We will refer to an iterator that returns pairs in this order as an *LRpairIterator*, and one that returns the same pairs but in the reverse order as a *RLpairIterator*.

Algorithms can be described by the *GenericSolve*() template instantiated with different iterator/heuristic function pairs I and H. For example, given an algorithm that uses the *LRpairIterator*, we can often substitute this with an *RLpairIterator*, which iterates through the same values that a *LRpairIterator* goes through, but emitted in reverse order. However, for the next-action problem, we claim that an *LRpairIterator* is preferable to an *RLpairIterator* because the variables $V_1,\ldots,V_n$ are interpreted as holding goals that an agent will try to achieve in order from V_1 to V_n. When required to act, the agent will always try to achieve the goal in V_1 first, and then possibly the goals in later variables. Thus it makes sense that an agent should spend more of its time "thinking" about what should go into V_1, and only worry about $V_2,\ldots,V_n$ once V_1 is reasonably satisfied. Indeed, the agent can act upon V_1 as soon as it is filled, and in discourse we can exploit this fact by having the agent issue a stalling phrase, such as *hmmm* or *just a moment*, in an effort to buy more time to think about satisfying V_1 (Section 5.4.3).

Along with this left-to-right property, algorithms that use the *LRpairIterator* can also

make guarantees about when they will no longer change the value of a variable. Consider again the loop form for *LRpairIterator* above. The outer loop steps i from 1 to $k-1$, while the inner loop steps j from $i+1$ to k. Thus, if *LRpairIterator* has just emitted (a, b), then for all future values (c, d) that it will emit, $c \geq a$ and $d > a$. Assuming no side-effects due to calling H, if (a, b) has just been emitted, we can mark all the variables before the one containing a and know they will not change unless goals are altered or the penalty matrix changes.

As shown in Figure 4.11, each of the algorithms listed below also has an *iterated* version where the basic algorithm is run repeatedly until the penalty no longer decreases. In the following sections, we will describe the various local search algorithms tested on the next-action problem.

4.6.1 Greedy

The Greedy algorithm looks at variables in left to right order, and swaps the value of the current variable V_i with the value in the variable after V_i that results in the greatest penalty decrease. If no swap decreases the penalty, then none is made, and any ties are broken randomly.

```
procedure Greedy()
  for i ← 1 to k do
    greedySwap(i)
  end for
end Greedy

procedure greedySwap(i)
  swap V_i with the variable in V_1, ..., V_k that decreases the penalty the most
end greedySwap
```

4.6.2 Left-right Greedy

This is the same as the Greedy algorithm, except the best value for variable i is chosen only by examining variables to the right, i.e. $i+1, \ldots, k$ instead of $1, \ldots, k$.

procedure LeftRightGreedy()
  for $i \leftarrow 1$ **to** k **do**
    leftRightGreedySwap(i)
  end for
end *LeftRightGreedy*

procedure leftRightGreedySwap(i)
  swap V_i with the variable in V_{i+1}, ..., V_k that decreases the penalty the most
end *leftRightGreedySwap*

4.6.3 Choose Two

I and H are defined to work with the *GenericSolve*() template in Figure 4.11.

I=LRpairIterator

procedure H(V_i,V_j)
  if swapping V_i and V_j decreases the overall penalty score **then**
    return true
  else
    return false
  end H

4.6.4 Two Opt

Two Opt uses the same heuristic and iterator as Choose Two, except instead of swapping the values of V_i and V_j in the generic solver of Figure 4.11, the values in the subsequence of variables from V_i to V_j (inclusive) are reversed.

Two Opt cannot be written in iterator/heuristic form like Choose Two because instead of swapping two elements, it requires that a whole sub-sequence be reversed; the template in Figure 4.11 could be modified to include Two Opt by passing an "action" function F as a parameter, and replacing the line that swaps the variables with a call to F, e.g.

procedure MoreGenericSolve(I,H,F)
  while *I.hasMoreElements*() **do**
    (V_i,V_j)$\leftarrow$*I.nextElement*()
    if H(V_i,V_j) is **true** **then**
      do F(V_i,V_j)
    end if
  end while
end *MoreGenericSolve*

Now Two Opt can be defined as I=*LRpairIterator*, H the same heuristic as for Choose Two, and F=*reverse*.

4.6.5 Random Choose Two

The same as Choose Two, except some "random walking" is allowed by sometimes swapping variable values that do not cause any change in the overall penalty.

I=*LRpairIterator*

procedure H(V_i,V_j)
  if swapping V_i and V_j decreases the overall penalty score **then**

return true

elsif swapping V_i and V_j does not change the penalty score **then**

return true with 50% probability, otherwise **return false**

else

return false

end if

end H

4.6.6 Sort Opt

Here, $p(i, j)$ is the ij-th entry of the penalty matrix. Sort Opt only looks at p, and does not sum any penalty values, so it is more efficient than other algorithms with the same iterator.

I=*LRpairIterator*

procedure H(V_i,V_j)

return $p(i,j) > p(j,i)$

end H

4.6.7 Single Sort Pass

This is similar to Sort Opt, except that a different iterator is used, i.e. it checks the $k-1$ adjacent variable pairs.

I=$<(1,2),(2,3),\ldots,(i,i+1),\ldots,(k-1,k)>$

procedure H(V_i,V_j)

return $p(i,j) > p(j,i)$

end H

4.6.8 Pre-process

The idea behind this algorithm is that if row i of the penalty matrix has a small sum, then its corresponding goal a_i should probably appear later in an ordering because later-appearing goals may add more to the final penalty total.

```
procedure H(V_i,V_j)
  ProcessPenaltyMatrix() /* NEED ONLY BE CALLED ONCE FOR EACH PENALTY
MATRIX p */
  return Pri[i] > Pri[j]
end H

procedure ProcessPenaltyMatrix()
  for i ← 1 to size(p) do
    Pri[i]←sum of ith row of p
  end for
end ProcessPenaltyMatrix
```

4.6.9 Results

The relative performance of the algorithms was essentially the same for the number of actions $t = 5, 10, \ldots, 50$, so we present some representative charts and graphs. On average, the most successful algorithms were the Greedy and Choose Two algorithms. As shown by the graph in Figure 4.12, where $t = 40$ actions were used, and the data in Table 4.1 and Table 4.2, where $t = 30$ actions were used, the iterated greedy algorithm is the overall winner, followed by the greedy method, and then a simple variation on this called iterated Left-right Greedy; the iterated version of an algorithm repeatedly applies the basic algorithm to the problem until the total penalty stops decreasing.

	Greedy	Left-right Greedy	Choose Two	Two Opt	Pre Process	Single Sort	Sort Opt
min	1.00	1.00	1.00	0.00	-0.89	0.00	-0.10
max	7.67	4.00	2.50	0.04	1.13	1.00	2.00
range	6.67	3.00	1.50	0.04	2.02	1.00	2.10
median	2.50	1.75	1.31	0.02	0.19	1.00	0.36
mean	2.58	1.83	1.35	0.02	0.19	0.98	0.41
var	0.65	0.20	0.05	0.00	0.06	0.02	0.07
std	0.80	0.45	0.23	0.01	0.24	0.14	0.27
iqr	1.00	0.61	0.28	0.01	0.29	0.00	0.31

Table 4.1: Statistics on per-swap average penalty decrement. The mean, for instance, represents the average penalty decrease resulting from each swap it performs. The higher the mean the better. In each case, 30 actions were used.

	Greedy	Left-right Greedy	Choose Two	Two Opt	Pre Process	Single Sort	Sort Opt
min	108.00	112.00	108.00	116.00	119.00	128.00	110.00
max	157.00	156.00	157.00	161.00	166.00	170.00	157.00
range	49.00	44.00	49.00	45.00	47.00	42.00	47.00
median	139.00	139.00	139.00	142.00	147.00	149.00	139.00
mean	138.07	138.93	138.40	141.74	146.93	148.71	138.56
var	42.99	39.89	43.29	45.86	56.56	46.59	51.82
std	6.56	6.32	6.58	6.77	7.52	6.83	7.20
iqr	9.00	8.00	9.00	9.00	10.00	9.00	10.00

Table 4.2: Statistics for final penalty score. The data here is the final penalty score of each test problem, so for the mean and min/max statistics, the lower the better. In each case, 30 actions were used.

While the iterated greedy method performed the best in terms of minimizing G, it is not necessarily the best choice for the action selection problem, at least for discourse. The problem is that at almost any time, the first action — the one that will be executed next if the agent is required to act — could be swapped with any of the later actions as the algorithm progresses. This can cause "unstable" behaviour, where the agent believes

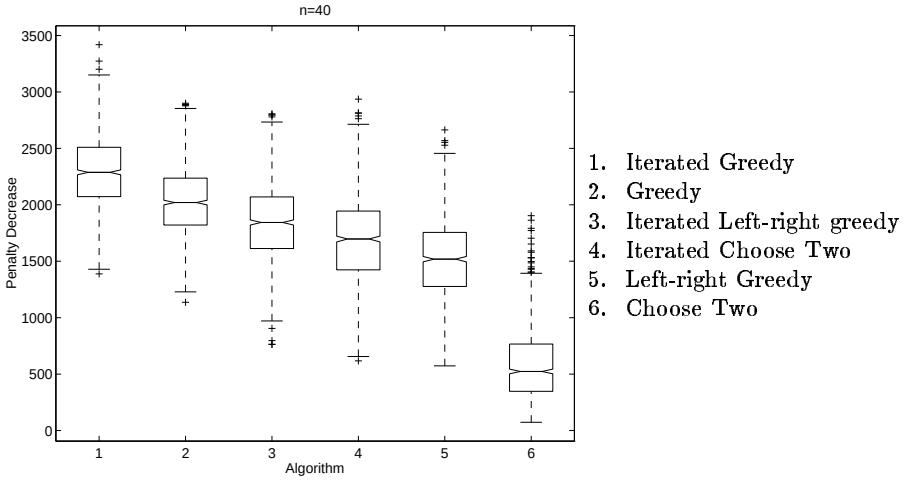


Figure 4.12: Boxplots for the top three classes of algorithms. The notch in the box represents the mean penalty decrease for the 1000 runs with 40 actions, while the ends of a box denote the lower and upper quartiles of the data; a + represents an outlier. The vertical axis indicates how much the algorithm decreased the penalty, so in this graph, higher is better. The rankings are based on the average performance of the algorithms.

for a while that, say, action a_i is the best choice, but then, near the end of its processing, swaps a_i with another action. Instead, we would prefer an algorithm that allots its processing time in a more segmented manner, so all the processing choosing the next action is done near the beginning, not distributed over the entire processing time as in the greedy algorithm. The advantage is that we can then know when a reasonable next action has been selected without waiting for the entire algorithm to finish; any remaining time can be spent ordering the remaining actions for the future.

So instead of the basic greedy algorithm, we prefer either Choose Two, or Left-right Greedy. Both the these algorithms can mark variables as having been filled, which can be used as a promise that the value of that variable will not change unless the action set or penalty function changes.

As shown in Table 4.1 and Table 4.2, the Pre-Process and Sort Opt algorithms will sometimes cause the overall penalty to increase. This is because they essentially derive a

heuristic from the penalty matrix which might not be always be a good one. Somewhat surprisingly, the Two Opt algorithm, which works well for the regular traveling salesman problem, performed quite poorly here. Intuitively, the reason would seem to be that Two Opt reverses subsequences of values, which is much more disruptive in the next-action problem because the scoring function depends upon the values of all the elements of the sequence, and not just the end-points of it as with the traveling-salesman problem.

4.7 Conclusion

This chapter has discussed the second step of the general turn-taking model: what should an agent say, and how should it decide among multiple goals/actions? We introduced the next-action problem, a novel way of reasoning about multiple goals based on the constraint-satisfaction paradigm. Further, we argued that local search algorithms are good ways to solve the next-action problem in a turn-taking framework because they are natural anytime algorithms that can work in dynamic domains with no or minimal changes (in contrast, for example, to backtracking style algorithms, which often assume a static environment and hard constraints), and presented some empirical results of a number of plausible local search algorithms.

In Chapter 6 we will discuss the *SG* system, which decides what to say by solving the next-action problem, and uses the local search algorithms of Section 4.6 to order and select its actions.

Chapter 5

When to Speak: Turn-taking Signals

> Good evening. Well tonight, we are going to talk about . . . that is . . . I am going to talk about . . . well actually I am talking about it now . . . well I'm not talking about it now, but I am talking . . . I know I'm pausing occasionally, and not talking during the pauses, but the pauses are part of the whole process of talking . . . when one talks one has to pause . . . er . . . like then! I paused . . . but I was still talking . . . and again there! No the real point of what I'm saying is that when I appear not to be talking don't go nipping out to the kitchen, putting the kettle on . . . buttering scones . . . or getting crumbs and bits of food out of those round brown straw mats that the teapot goes on . . . because in all probability I'm still talking and what you heard was a pause . . . er . . . like there again. Look! To make it absolutely easier, so there's no problem at all, what I'll do, I'll give you some kind of sign, like this (makes a gesture) while I'm still talking, and only pausing in between words . . . and when I've finished altogether I'll do this. All right?
>
> *Mr. Orbiter, Monty Python's Flying Circus*

5.1 Introduction

This chapter discusses the concept of turn-taking, both in general and with respect to turn-taking in conversation. Section 5.2 discusses turn-taking in multi-agent systems, and describes a state-based model of turn-taking that identifies taking a turn as an

agent making a decision in a particular state. This section also introduces the notion of initiative, and suggests how initiative and turn-taking might fit together. Section 5.3 next presents a formal description of turn-taking based on the situation calculus. With it, one can model turn-taking situations as a "turn object", such as the conversational "floor", being shared and passed around by multiple agents. This formal model provides an omniscient, third-party view of the interaction, and so in Section 5.4 we discuss how a single agent can take its turn in a conversation.

5.2 Taking Turns

Turn-taking is a very general concept. As mentioned in Section 1.1, humans begin to practice conversational turn-taking as infants, and this lifetime of practice pays off for most people, who take turns in conversations with relative ease and often little conscious effort. The logic presented in Section 5.3 is a step towards formalizing conversational turn-taking, in an effort to enable computers to converse in a natural way with humans. However, while conversation provides the most obvious example of a *turn-taking system*, there are many other instances where people take turns, and the turn-taking logic can also model many of these situations. In the following sections, turn-taking will be analyzed at a very general level, and conversational turn-taking will be seen as just one particular instantiation of this general model. We begin with an example.

Taking Turns Getting Lost

Madge and Trygve are lost in Soho. They stop at an intersection and must decide which way to go, so Trygve makes a suggestion:

Dialog 5.1

Trygve: I think we should turn left.

Madge: Keen!

They turn left, and are still lost. After a minute Trygve makes another suggestion:

Dialog 5.2

Trygve: Let's go down a block and try turning left.

Madge: Yeah!

They do this and still end up lost. Once again, they stand thinking about what to do next and Trygve says:

Dialog 5.3

Trygve: I'm not having much luck. Which way do you think we should go?

Madge: Alright, let's try this way.

In each of the three examples, which we treat as three separate dialogs, Trygve has taken the initiative to speak first. Starting any conversation requires, at least, that one agent make a decision to begin speaking, and we will refer to this decision as the agent *taking the initiative to speak*, or, more precisely, taking the initiative to take a speaking turn. In the examples, it is possible that Madge might start to speak at the same time as Trygve. In that case, a floor-battle would result, and this conflict for the conversational floor would need to be resolved (Section 5.3.3). In Dialog 5.1 and Dialog 5.2, Trygve also takes the lead in deciding which street to turn down, but in Dialog 5.3, he instead asks Madge to navigate. Along with taking the initiative to speak, Trygve has taken the initiative in offering Madge the responsibility for deciding which street to walk down.

Note that these are two separate instances of initiative-taking. Madge accepts the responsibility to make the next "direction-decision", but at no time does she take the initiative to speak or to make a direction-decision. Because only Trygve has performed initiative-taking actions, we can say that Trygve is both in control of the conversation, and in control of the direction-decisions. Suppose the following dialog occurred in place of Dialog 5.3:

Dialog 5.4

Madge: Let me navigate. Let's try this left here.

Trygve: Okay.

Here, Madge has taken both the initiative to speak, and the direction-decision initiative. It would also be possible for Madge to take just the initiative to speak, e.g.:

Dialog 5.5

Madge: Now what?

Trygve: How about left here?

Or she could just take the direction-decision initiative, e.g.:

Dialog 5.6

Trygve: I'm not sure where we are, let me see ...

Madge: Let's try going left.

At least two things are going on: Madge and Trygve take turns speaking in the conversation, and also take turns in making direction-decisions. These are separate processes, but they are closely related here since Madge and Trygve are using conversation to coordinate their interaction.

Turn-taking decisions are usually made by a single agent, but collaboration is possible. A group of agents might negotiate and come to an agreement about what the content of a turn should be, but there must be an agreement to act together, as if the pair were a single agent making the decision. For example, Madge and Trygve might become more collaborative in their direction-decisions:

Dialog 5.7

Madge: I don't know, do you think we should turn left again?

Trygve: Yeah, let's try turning left.

Here, Madge seeks and gets agreement on her suggestion to turn left. While this is a collaboration, it makes sense to say that Madge is responsible for the direction-decision since she is the initiator and is essentially asking Trygve to respond yes or no. Even if Trygve does not agree with her suggestion, Madge can still keep responsibility for the direction-decision being discussed, e.g.:

Dialog 5.8

Madge: I don't know, do you think we should turn left again?

Trygve: We've already been that way.

Madge: Okay, how about straight and then left?

Trygve: Let's try that.

Of course, Madge could give, or Trygve could take, responsibility for the direction-decision, e.g.:

Dialog 5.9

Madge: I don't know, do you think we should turn left again?

Trygve: We've already been that way.

Madge: Okay, how about straight and then left?

Trygve: I don't know about that.

Madge: Hmmm... [shrugs]

Trygve: Alright, let's go back and turn right.

Madge: Sure.

In this dialog, when Madge shrugs she drops her responsibility for making the the direction-decision that she possessed previously. At that point, either she or Trygve could pick up that responsibility, and in this case Trygve does by making the next direction-decision (which Madge accepts). Clearly, the responsibility for the direction-decisions could be negotiated back and forth in this manner indefinitely.

Turn-taking in the Abstract

The examples in the previous section suggest a very general way of characterizing turn-taking, and some kinds of initiative. In this section, these ideas will be generalized, and then in Section 5.3 a subset of them intended mainly for, but not limited to, conversational turn-taking will be formalized.

A *turn-taking system* is defined in terms of a state-based model of decision making. In this model, one imagines the space of possible decisions to be a graph, where nodes represent world states, and edges represent all the decisions that can be made in the state from which they emanate. In a single-agent system, one can imagine the agent moving

from state to state, being in exactly one state at any one time. In a multi-agent system, we imagine a set of agents going from state to state, where again each agent is in exactly one state at any one time. The set of agents should be in the same state, but in general, each agent will have a personal representation of the state-graph in their own head. This can result in "coordination problems", where agent's beliefs about the shape of, or their location in, the state-graph might be in conflict with the beliefs of other agents (or the actualities of the real world). As in any situation relying on mutual information, agents must be prepared to deal with inconsistent beliefs.

For simplicity, we will only deal with the case where two agents, A and B, who are traversing a state-graph together. We assume that they have no coordination problems, which is a reasonable assumption for most turn-taking systems, especially conversational ones since it is usually clear to everyone who has the conversational floor (i.e. usually the person speaking). Suppose the agents are both in state S, and the edges emanating from S correspond to the possible decisions, $\{d_1, \ldots, d_n\}$, that they can make in S. How do they choose which edge to follow, i.e. which decision to make? Clearly, to make sure they move to the same next state, A and B must somehow communicate and agree upon which decision to make. We require that exactly one of A or B make the choice, but how they make it is not specified, and is left up to them and their situation. They might explicitly collaborate on decisions, or follow some pre-set protocol, or follow any combination of methods. The procedure that the agents repeatedly follow is then this:

- A and B are in state S and must choose exactly one of $\{d_1, \ldots, d_n\}$ to follow to the next state;
- A and B, between themselves, choose one of them to have the responsibility for selecting from $\{d_1, \ldots, d_n\}$; the nominated agent is said to have the turn.

After A and B together make a move to a new state, the state just vacated can be labelled

according to who had the decision-making turn in that state; we call this the state's *turn label*. If A and B travel through the state graph starting at node S and ending at node S', then for each node in the path between S and S', we can record whether it was A or B who was responsible for choosing the next node to go to. In general, there is no restriction on which agent can make a turn in a given state, although individual turn-taking systems might stipulate certain constraints. For example, in conversational turn-taking, it is required that the turn labels between S and S' alternate, either $ABAB\ldots$ or $BABA\ldots$. This is because if one speaker makes more than one utterance in a row, all of the utterances are counted as being part of the same turn. In the direction-decision example of Section 5.2, the decisions in a state are the set of possible directions that Madge and Trygve can take, and it is possible for any string of turn labels to result. For instance, if Madge is an expert on the geography of Cambridge, England, then she might make every direction-decision there. If Trygve is the geography expert in Cambridge, Ontario, then he might make every direction-decision there. In Soho, where neither Madge nor Trygve have been before, they might alternate their decision-direction turns, or make them in an ad hoc or collaborative fashion as suggested by the examples in Section 5.2.

Initiative

The amount of knowledge that agents share influences *initiative*. As already mentioned, a simple example of taking the initiative occurs when an agent begins a conversation, which we refer to as taking the initiative to speak. There are many reasons for initiating a conversation: the agent might want information, they might want help performing some task, or they might just want to make small-talk. Typically, we will assume agents are initiating some kind of advising or task-oriented dialog, and so will usually have some concrete reason related to the domain for starting a dialog. For example, Smith et al. (1995) (discussed more in Section 7.6) treat conversational agents as theorem-provers,

and such agents initiate a dialog when they believe they lack the information needed to prove their theorems. This desire to find the "missing axioms" also motivates all of the other initiative decisions the agent makes throughout the dialog.

Figure 5.1 explains the difference between intelligent advising and intelligent collaboration systems. Initiative becomes most interesting in *mixed-initiative* systems, where who knows what is initially unknown. Mixed-initiative systems typically allow for the possibility that, for any particular proposition, it is unknown if both agents know or do not know it, or if only one of them knows it; non-mixed-initiative systems usually constrain who knows what ahead of time. Generally, if an agent realizes it knows some relevant fact that the other agent does not know, then, all other things being equal, it ought to take the initiative and communicate this knowledge. This entails at least taking a turn in the dialog, and possibly taking a turn in other turn-taking systems that the agents are part of. With respect to conversational turn-taking, taking the initiative will mean that the agent wants to make a particular utterance, hence it must get control of the conversational floor if it does not already have it.

An agent that never takes the initiative will only take turns in reaction to the turns made by the other agent. For example, if A asks B a question, then normally B will try to answer A's question. It might be understood that, in answering the question, B can ask for clarifications and such, but its actions at this point are limited by its commitment to answering A's question. In a mixed-initiative system, however, it could happen that B decides it has something more important than answering A's question to say, and so responds with a question of its own, or even a digression or return to a previous topic. In a mixed-initiative system, the commitment to solve the overall domain problem will sometimes take precedence over the commitment to answer a particular question or to achieve any particular sub-goal.

Defining initiative is surprisingly tricky, and while there is general agreement on the

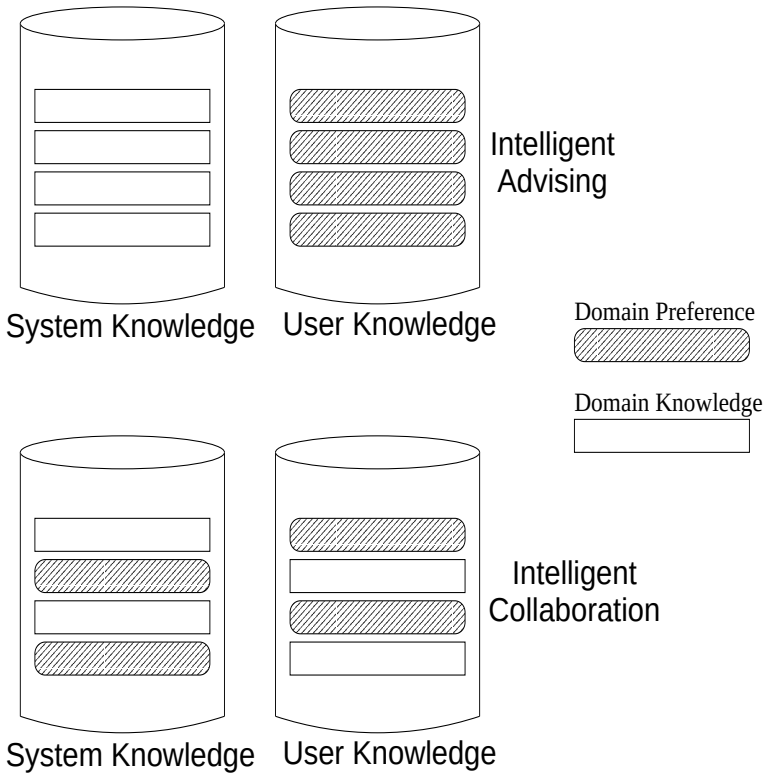


Figure 5.1: Intelligent advising versus intelligent collaboration. The top picture shows an intelligent advisor: the system has all the knowledge about the domain, and the user has all the preferences. Both the system and the user know that the system has all the domain knowledge (and is the expert), and that the user is the information-seeker who knows their preferences but has no substantial domain knowledge. A characteristic of such an arrangement is that one should be able to get a correct answer from the system to any question about the domain because it has all the domain knowledge. The second picture shows an intelligent collaboration system: both the system and the user have some knowledge of the domain and some preferences. In this case, the union of the user's knowledge and the system's knowledge can include the entire domain model, or just a part of it. At the start, at least, the system and user are not certain what knowledge the other agent has (or what actions the other is capable of), and so exchanges concerning "who knows what" can occur, in contrast to the intelligent advising case where such questions are superfluous. Mixed-initiative systems arise mainly in the context of intelligent collaboration.

core intuitions, there is no consensus on the finer details of initiative, which are usually different for different researchers (Haller and McRoy 1997a). There appear to be some basic reasons why there is not yet a commonly accepted definition of initiative. First, initiative is an abstract concept, in contrast to conversational turns, which are comparatively easy to identify (i.e. the agent who is speaking usually has the floor). There is, also, no agreement on the ontological status of initiative: sometimes initiative/mixed-initiative, is spoken of as a property of a system/agent/algorithm, and sometimes initiative/mixed-initiative is spoken of as an abstract entity, like the conversational floor, which agents can take, drop, etc. It's unclear if initiative is something that can come in degrees, or is a binary concept, if it is an emergent property of a system, or a fundamental mechanism that must be explicitly programmed. Without a common model for thinking about initiative, it is not surprising that there is no agreement on what it means.

But even if we settle on a common model, such as the state-graph model outlined in Section 5.2, there are still subtleties. The example dialogs in Section 5.2 illustrate (at least) four interrelated things: conversational turn-taking, conversational initiative, direction-decision turn-taking, and direction-decision initiative. What can be confusing is that instances of these phenomena frequently co-occur. For instance, we can use these four phenomena in an explanation of how Trygve comes to make his utterance in Dialog 5.1. First, presumably Trygve has an idea about how to get out of Soho, and he decides to share this information with Madge; this is Trygve taking the direction-decision initiative. This causes Trygve to take the initiative to speak, which then causes Trygve to take a speaking turn and say *I think we should turn left.* After Madge responds with *Keen!* and they agree to turn left, Trygve has taken his turn making a direction-decision. The sequence of events is this:

direction-decision initiative → conversational initiative → conversational turn → direction-decision turn.

What is still unclear is if this chain of events is just a *description* of the behaviour of an intelligent agent, or if it is a description of the explicit reasoning that an agent must do.[1] It is not clear if an agent ever needs to reason explicitly about initiative, e.g. "If I take the initiative now, then X will happen; but if I don't take the initiative, then Y will happen." It does seem that people do *some* explicit reasoning about turn-taking initiative, as evidenced by instances of people deciding whether they should tell someone some potentially unpleasant fact, or when a listener weighs the pros and cons of interrupting a speaker to provide a new piece of information. While a general theory of initiative and turn-taking would certainly include such examples, it is not clear if these examples cannot just be handled as special cases of standard methods.

Other researchers have noted a similar distinction between kinds of initiative. For example, Chu-Carroll and Brown (1997) distinguish between *task* and *dialog* initiative. An agent is said to have the task initiative if it directly proposes actions to be performed, and it is said to have the dialog initiative when it tries to establish a mutual belief. Task initiative is a subset of dialog initiative in the sense that proposing an action entails an attempt to establish the mutual belief that the action be adopted.[2] Other researchers have made similar distinctions (Haller and McRoy 1997b), although there is not yet a

[1] If initiative causes turn-taking, then what causes initiative? An agent's perceptions of the world will change its mental states, and it is ultimately particular changes in mental state that cause an agent to take the initiative.

[2] These definitions are far from a satisfying theory of initiative, though. For example, it seems possible for an agent to take the task initiative by simply performing an action without the consent or even knowledge of the other agent. For instance, one of two people moving a large cabinet might adjust their grip on the cabinet in such a way to prevent the doors from flying open when they move it. Intuitively, the mover has taken some sort of initiative and performed an action that influences the entire collaboration. Since they have not proposed this action, then it would not count as an instance of task-initiative according to Chu-Carroll and Brown's definition. The problem is that Chu-Carroll and Brown tie task-initiative directly to dialog.

"grand unifying theory" that explains initiative, mixed-initiative, turn-taking, and their relations to other aspects of multi-agent interaction.

5.3 Introduction to the Logic of Turn-taking

In this section, we give a formal description of turn-taking that allows agents to make and reject offers and requests for the "floor". A basic fact of multi-party conversation is that only one agent can have the floor at any one time, while everyone else must be a listener. Research in multi-party discourse has usually taken turn-taking to be straightforward and obvious, although as argued in Section 5.1 part of the difficulty in defining more complex notions, such as initiative, stems in part from an imprecise understanding of turn-taking. With a clear understanding of turn-taking, other complex features of interaction can be better delineated and understood. In addition, a precise account of turn-taking aids in the specification and design of dialog systems (Chapter 6).

The logical formalism for turn-taking is based on the situation calculus. In its most general rendering, we consider a set of objects called *agents* that are able to possess a single indivisible object X (the "floor"). One of the agents, called N (i.e. "nature"), is special and accounts for "acts of nature". Agent N has the same abilities as other agents. For example, suppose the agents are basketball players, and X is the basketball. At most one player can have the basketball, and they can drop it, hold on to it if someone else tries to steal it, or offer it to someone else (e.g. by making a motion to pass them the ball). Players without the ball can try to take it, or request to be given it. Players can fight over possession, and the winner is usually the one who is the strongest or most agile. If the ball is still in the court, and not possessed by anyone (e.g. it is rolling on the ground), we say that nobody has the ball. If the ball leaves the court, then we can say that N has the ball. To avoid explicitly representing all the exceptional situations

that might occur, we imagine that N has taken possession of the ball whenever anything unexpected happens.

Another example is of a person, a microwave oven, and a chicken pie X. The person "gives" the pie to the oven, which signals to the person when it is cooked. If the person is not careful, N can take the pie from them (e.g. it gets dropped on the floor). In all of the examples we want to model, a group of agents are concerned with passing a single object among themselves. While the particular scenario dictates what actions are useful and possible for the agents to perform, actions related to possession of X are clearly the most relevant ones. This chapter defines a logical formalism geared towards dialog, where X is the conversational floor, the agents are the conversants, and N is treated as a force that can cause unexpected movements of X.

5.3.1 The Situation Calculus

We use the *situation calculus*, a variety of predicate calculus useful for describing and reasoning about a changing world (McCarthy and Hayes 1969; Reiter 1991; Levesque et al. 1997). The situation calculus combines first-order logic with actions, and, intuitively, a *situation* is a logical description of a state of the world. An *action* encodes transitions between situations, and consists of a set of *pre-conditions* that must be true for the action to be possible in a situation, and a set of *effects* that describe how the situation changes if the action is successfully performed. The notation $do(\alpha, s)$ denotes the *successor situation* that results from performing action α in situation s.[3] When we talk about an action α being possible, we write $Poss(\alpha, s)$ which, for our purposes, will be true exactly when α's pre-conditions hold. Predicates that take a situation as a parameter are called *fluents*,

[3]For clarity we will use a STRIPS-like notation for defining actions although they could all be defined using just logical sentences.

and by convention the situation is the last argument of a fluent.[4] We also follow the convention that all free variables are universally quantified from the outside.

Consider a simple example involving $Madge$ and a piece of $Cheese$. We define the following:

- $\mathit{hungry}(p, s)$ is a fluent that holds if p is hungry in situation s;
- $\mathit{edible}(x, s)$ is a fluent that holds if x is edible in situation s;
- $\text{NIBBLE}(x, y)$ is a parameterized action, its precondition is $\mathit{hungry}(x, s) \wedge \mathit{edible}(y, s)$, and its effect is $\neg\mathit{hungry}(x, \mathit{do}(\text{NIBBLE}(x, y), s))$, i.e. that x is no longer hungry in the state that results from performing $\text{NIBBLE}(x, y)$ in situation s.

In the initial situation S_0, the cheese is edible and Madge is hungry, so $\mathit{edible}(Cheese, S_0)$ and $\mathit{hungry}(Madge, S_0)$ are true. The action $\text{NIBBLE}(Madge, Cheese)$ is possible in S_0, and if the action is performed the resulting situation is $S_1 = \mathit{do}(\text{NIBBLE}(Madge, Cheese), S_0)$, in which $Madge$ is no longer hungry. Intuitively, any remaining cheese should still be edible, but nothing in the given domain model allows this fact to be derived. It is necessary to add *frame axioms* such as

$$\mathit{edible}(Cheese, s) \supset \mathit{edible}(Cheese, \mathit{do}(\text{NIBBLE}(Madge, Cheese), s)).$$

The *frame problem*, originally described by McCarthy and Hayes (1969), is the general problem of usefully specifying how fluents change from situation to situation. Explicitly writing frame axioms is unwieldy because, in general, in a domain with a actions and e fluents, $O(ae)$ frame axioms will be needed. Fairly general and economical solutions to the frame problem in the situation calculus do exist (e.g. Reiter 1991; Elkan 1992), although

[4] Some authors use a special *holds* operator, and so write $\mathit{holds}(\mathit{pred}, s)$ to mean predicate *pred* is true in situation s.

the domains dealt with in this thesis are small enough that we can just use (assume) all the necessary frame axioms; for more complicated domains, a more parsimonious solution can be used.

5.3.2 Multi-agent Single Object Logic

Using the situation calculus, we define a universe of objects consisting of *agents* and a single unique object X.[5] There is one special agent N, otherwise known as "Nature", which is held responsible for unexpected actions, i.e. "acts of nature".[6] Using N, we can stipulate that in any given situation the successor situation is assumed to change only due to actions of agents, and still allow for some kinds of uncertainty caused by N. N is capable of exactly the same actions as other agents, although N's actions need not be part of any rational plan, e.g. N could act randomly.

We use the letters p, q, r to refer to an agent. We assume that the set of agents and the particular object X are designated at the beginning, and stay fixed, i.e. no agents come or go, and X is always the same indivisible object.

In any one situation, an agent must perform at least one action (which can be the empty "do nothing" action), and can perform at most one single-object logic action; when discussing utterances, we will introduce actions that can be done at the same time as a single-object logic action. If there are $m+1$ agents, $\{p_1, \ldots, p_m, \mathsf{N}\}$, then in any situation all $m+1$ agents will act, in parallel. Let $\vec{\alpha} = <\alpha_{p_1}, \ldots, \alpha_{p_m}, \alpha_{\mathsf{N}}>$ be the vector of the $m+1$ actions performed in a given situation. An action vector $\vec{\alpha}$ is possible in a situation if and only if each of its component actions is possible, e.g. $\mathit{Poss}(\vec{\alpha}, s) \equiv \mathit{Poss}(\alpha_{p_1}, s) \wedge \ldots \wedge \mathit{Poss}(\alpha_{p_m}, s) \wedge \mathit{Poss}(\alpha_{\mathsf{N}}, s)$. After all these actions have

[5] X could conceivably be one of the agents, but, for our purposes, we will not allow that possibility.

[6] The idea of treating nature as an agent with the same abilities as the other agents comes from game theory.

been executed, and any conflicts have been resolved, we denote the resulting situation by $do(\vec{\alpha}, s)$. Since there is only one object X for the agents to act upon, conflicts among actions in $\vec{\alpha}$ are greatly simplified because actions can only conflict on the placement of X.

For example, let $\vec{\alpha}$ and $\vec{\beta}$ be action vectors. If $\vec{\alpha}$ is possible in S_0, doing it will result in the state $do(\vec{\alpha}, S_0)$, and, if $\vec{\beta}$ is then possible, the situation after doing $\vec{\beta}$ will be $do(\vec{\beta}, do(\vec{\alpha}, S_0))$. In general, it is impossible to know the resulting state ahead of time, because at least N's action is usually unknown, and the actual method for resolving conflicts is not specified within the logic itself.

Fluents

We define four fluents that describe the state of the world with respect to X and the agents.

The first keeps track of who has possession of X:

- $has(p, \mathsf{X}, s)$ means that p possesses X in situation s

The expression $\neg has(p, \mathsf{X}, S)$ means nobody has X in situation S. The next two fluents are for the offers and requests that can be made:

- $offer\text{-}pending(p, \mathsf{X}, q, s)$ means that an offer from p to q for X is pending in situation s

- $request\text{-}pending(q, \mathsf{X}, p, s)$ means a request from q to p for X is pending in situation s

Since X is the only object that an agent can have, we require that at most one agent can have X.

Axiom 1 *In any one situation, at most one agent can have* X.

$$\mathit{has}(p, \mathsf{X}, s) \wedge \mathit{has}(q, \mathsf{X}, s) \supset p = q$$

We require that these above fluents change their truth-values only due to actions that can affect them. This means, for example, that an agent who has X will continue to have X in the next situation if no action is performed by any agent that changes the *has* fluent. These requirements can be explicitly encoded as frame axioms.

The fourth and final fluent is $\mathit{win}(p, \mathsf{X}, s)$, and is true when p has won the *competition* for X that occurred in s. An action is *competing for* X if a possible effect of that action is for the doer to end up having X in the successor state. More specifically, a competition arises in a situation if $\vec{\alpha}$ contains two or more competitive actions, and $\mathit{win}(p, \mathsf{X}, \mathit{do}(\vec{\alpha}, s))$ is true if and only if p is the winner of the competition. A competition resolution procedure must be defined, and for practical purposes we do not specify this in logic, but assume such a procedure will be associated with the *win* fluent.

Next, we define all the actions that agents can perform. We refer to the agent with X as the *holder*, and agents without X as *non-holders*. As a notational convenience, we use the symbol $\triangleright$ to refer to the successor situation $\mathit{do}(\vec{\alpha}, s)$.

Holder Actions

NULL(p)

pre: *none*

effect: *none*

The null action can be done by any agent in any situation, and has no effects.

KEEP(p, X)

pre: $\mathit{has}(p, \mathsf{X}, s)$

effect: $\mathit{has}(p, \mathsf{X}, \triangleright)$ **if** $\mathit{win}(p, \mathsf{X}, s)$

An agent with X must make an effort to keep it if someone is trying to take it away.

DROP(p, X)

pre: $\mathit{has}(p, \mathsf{X}, s)$

effect: $\neg\mathit{has}(p, \mathsf{X}, \triangleright) \wedge \neg\mathit{offer\text{-}pending}(p, \mathsf{X}, q, \triangleright) \wedge \neg\mathit{request\text{-}pending}(r, \mathsf{X}, p, \triangleright)$

All offers made by p, and requests made to p, are deleted since they no longer make sense if p does not have X.

OFFER(p, X, q)

pre: $p \neq q \wedge \mathit{has}(p, \mathsf{X}, s)$

effect: $\mathit{offer\text{-}pending}(p, \mathsf{X}, q, \triangleright)$

An agent can make an offer even if one is currently pending.

RETRACT-OFFER(p, X, q)

pre: $p \neq q \wedge \mathit{has}(p, \mathsf{X}, s) \wedge \mathit{offer\text{-}pending}(p, \mathsf{X}, q, s)$

effect: $\neg\mathit{offer\text{-}pending}(p, \mathsf{X}, q, \triangleright)$

An agent with X can retract any offer it has made to another agent.

REJECT-REQUEST(p, X, q)

pre: $p \neq q \wedge \mathit{has}(p, \mathsf{X}, s) \wedge \mathit{request\text{-}pending}(q, \mathsf{X}, p, s)$

effect: $\neg\mathit{request\text{-}pending}(q, \mathsf{X}, p, \triangleright)$

An agent with X can reject any requests made to it.

Non-Holder Actions

TAKE(p, X, q)

pre: $p \neq q \wedge \mathit{has}(q, \mathsf{X}, s)$

effect: $[\mathit{has}(p, \mathsf{X}, \triangleright) \wedge \neg \mathit{offer\text{-}pending}(q, \mathsf{X}, r, \triangleright) \wedge \neg \mathit{request\text{-}pending}(r', \mathsf{X}, q, \triangleright)]$ **if** $\mathit{win}(p, \mathsf{X}, s)$

The effect of this action is conditional, hence a take action by p is unsuccessful if p does not win the competition for X.

PICK-UP(p, X)

pre: $\neg \mathit{has}(q, \mathsf{X}, s)$

effect: $\mathit{has}(p, \mathsf{X}, \triangleright)$ **if** $\mathit{win}(p, \mathsf{X}, s)$

More than one agent might try to pick up X, so the winner of the competition for X is the only one who successfully picks it up.

REQUEST(p, X, q)

pre: $p \neq q \wedge \mathit{has}(q, \mathsf{X}, s)$

effect: $\mathit{request\text{-}pending}(p, \mathsf{X}, q, \triangleright)$

RETRACT-REQUEST(p, X, q)

pre: $p \neq q \wedge \mathit{has}(q, \mathsf{X}, s) \wedge \mathit{request\text{-}pending}(p, \mathsf{X}, q, s)$

effect: $\neg \mathit{request\text{-}pending}(p, \mathsf{X}, q, \triangleright)$

REJECT-OFFER(p, X, q)

pre: $p \neq q \wedge \mathit{offer\text{-}pending}(q, \mathsf{X}, p, s)$

effect: $\neg \mathit{offer\text{-}pending}(q, \mathsf{X}, p, \triangleright)$

These last three actions are similar to the last three holder actions.

From these definitions, we can derive $\mathit{request\text{-}pending}(p, \mathsf{X}, q, s) \supset \mathit{has}(q, \mathsf{X}, s)$. To see this for *request-pending*, note that a *request-pending* fluent can be set true only

by a REQUEST action, and that some agent q has X is a pre-condition of a REQUEST. The only actions that can cause agent q to not have X are DROP, REQUEST, or TAKE, each of which delete all pending requests made to q. Therefore, it is not possible for *request-pending*(p, X, q, s) to be true while *has*(q, X, s) is false. A similar argument shows *offer-pending*$(q, \mathsf{X}, p, s) \supset$ *has*(q, X, s).

5.3.3 Competing Actions

In general, two actions α and β *compete* if an effect of α is p, an effect of β is q, and $p \wedge q \supset \mathit{false}$. Since situations must be consistent descriptions of the world, α and β cannot both be successfully executed in the same situation. This is a basic problem that must be addressed in any multi-agent planning environment. This problem is similar to resolving threats in partial order planning (Weld 1994), and the particular factors relevant to resolving conflicts depend on the domain. For example, a competition for a basketball might be determined by strength, while competition for the attention of a cat might be determined by who has the tastiest food. For general approaches to multi-agent environments with parallel actions, see, for example, Georgeff (1984) or Giacomo et al. (1997). Factors to consider in defining a conflict resolution procedure for the conversational floor are discussed in Section 5.3.5.

5.3.4 Examples of Turn-taking Systems

An Intersection

Consider an ordinary four-way intersection with traffic lights in the center controlling the flow of traffic. East-west and north-south have their own traffic signal that at any one time displays either green (go), amber (slow down), or red (stop). The fundamental law of traffic intersections is: *at any time, at least one traffic light must be red.* This rule helps

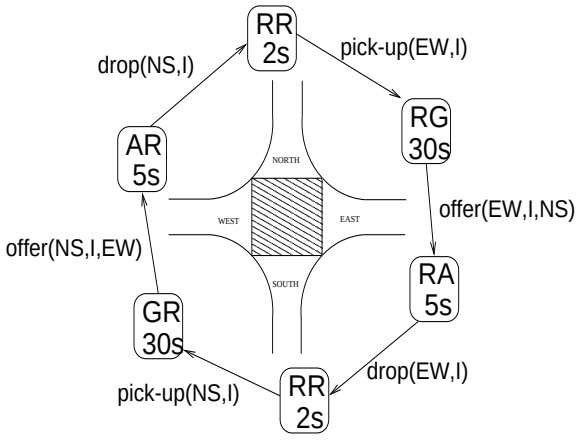


Figure 5.2: A 4-way intersection. Cars can go either east-west or north-south, but not both directions at the same time since they must share the (shaded) intersection. The signal changes are shown as a state transition diagram; for example, RA,5s means to keep the north-south light red and the east-west light amber for five seconds, and then to move to the next state, where both lights are kept red for two seconds. This process cycles indefinitely.

prevents cars from crashing into each other and guarantees that traffic flows east-west, north-south or not at all (assuming the drivers obey the signals!). The typical order of light-changes is shown in Figure 5.2.

Another way to think of a traffic intersection is as a system of consisting of 3 entities: an agent named East-west, an agent named North-south, and an Intersection object that they share. Either agent can possess, or hold onto, the Intersection, and so if East-west possesses the Intersection, then traffic can flow from east to west. The fundamental law of traffic intersections becomes: *at any one time, at most one agent can possess the Intersection.* Transitions between states are labelled, as in Figure 5.2, as offers, drops, or pick-ups of the intersection. Notice that a pick-up of the intersection corresponds to a light changing from red to green, an offer corresponds to a change from green to amber, and a drop of the intersection is a change from amber to red.

Figure 5.3 shows the transition diagram of Figure 5.2 "factored" into one diagram

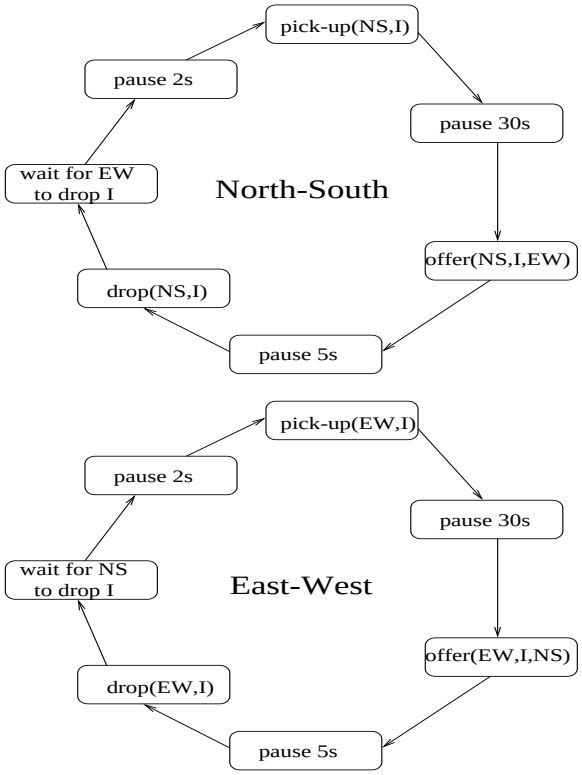


Figure 5.3: Transition diagrams for individual traffic control agents.

for each agent. In this example the offers that the agent makes are not used by the other agent; instead they are necessary as signals of impending changes to the cars that travel through the intersection. Table 5.1 shows how the actions map onto the logical actions.

The design of the system makes conflicting actions impossible. Of course, cars can (literally!) run into conflicts in the intersection if drivers disobey the signals. The idea of one object per location, which the traffic lights attempt to enforce in the intersection, is related to the fundamental physical fact that two objects cannot share the same location at the same time without usually causing damage to themselves. People, for example, cannot share the same physical space at the same time; if two people run headlong into each other, then the result will be painful for both of them. Following a turn-taking

Domain Actions	Holder Actions (p)	Non-holder Actions (q)	Domain Actions
	KEEP(p, I)	TAKE(q, I, p)	
amber to red	DROP(p, I)	PICK-UP(q, I)	red to green
green to amber	OFFER(p, I, q)	REQUEST(q, I, p)	
	RETRACT-OFFER(p, I, q)	RETRACT-REQUEST(q, I, p)	
	REJECT-REQUEST(p, I, q)	REJECT-OFFER(q, I, p)	

Table 5.1: An Intersection as a turn-taking System.

Domain Actions	Holder Actions (p)	Non-holder Actions (q)	Domain Actions
control ball	KEEP(p, I)	TAKE(q, I, p)	challenge ball carrier
pass, shoot kick	DROP(p, I)	PICK-UP(q, I)	gain control of free ball
	OFFER(p, I, q)	REQUEST(q, I, p)	call for ball
	RETRACT-OFFER(p, I, q)	RETRACT-REQUEST(q, I, p)	
	REJECT-REQUEST(p, I, q)	REJECT-OFFER(q, I, p)	

Table 5.2: Soccer as a turn-taking system.

protocol is a good way of avoiding conflicts.

This example shows the generality of the multi-agent single-object logic. However, there is little practical advantage to characterizing traffic lights as a multi-agent system in this way since the resulting agents are not autonomous, and the rules for how traffic enters the intersection are completely specified and known to everyone in advance.

A Soccer Match

A more complicated example of what the single-object logic can model is a soccer game. Here, the agents are the soccer players, and the ball is the single object. Nature, who is not counted as a player, also plays a role in cases were the ball leaves the playing field. See Table 5.2.

The player with control of the ball is said to have it; if the ball is high in the air or rolling untouched in the field, we say that nobody has it. A player with the ball can only lose it by kicking it away or losing a challenge made by another player. A player drops the ball if they pass it, kick it, shoot it, or otherwise lose control of the ball without a challenge by another player; see Table 5.2. Signals for passing/kicking the ball can be implicit or explicit. An implicit signal might be when the ball-carrier is in front of her opponent's net and makes motions that indicates she will shoot the ball. An explicit signal could be, for example, the ball-carrier pointing or calling to the person they want to pass the ball to. Players without the ball can request the ball, by shouting to the ball-carrier to pass it to them.[7] Practically, explicit requests and offers for the ball are not frequent in soccer since such information could be quite helpful to the opposition; games such as soccer are an example of a domain where agents likely need to understand and engage in deception to perform well (Donaldson 1994).

Competitions, or challenges, between individual players for the ball are frequent in soccer. Chance plays a major role in the outcome of any such competition, as does the speed, strength, and ball-handling skills of the competitors. A competition for the ball could be determined by calculating a weighted averaged of the relevant features for each competitor, and awarding possession to the player with the highest score.

This logic could be used as the basis for an annotation language describing the action

[7]Usually only team members of the ball carrier request the ball this way, but nothing except sportsmanship stops the other team from confusing the ball-carrier with requests for the ball.

that occurs during a soccer game. More specific fluents and actions would be required to describe particular kinds of passes, shots, stoppages in play, etc. Such a language could also be used for planning strategies for soccer robots,[8] although whether logic could be used to guide real-time soccer robots in something like the RoboCup would be a difficult challenge (Kitano et al. 1997).

5.3.5 The Logic of Turn-taking

The logic of turn-taking is built from single-object logic: X is the conversational floor, and the agents (other than N) are the discourse participants. Table 5.3 shows how the concepts of the turn-taking domain map to the actions of the turn-taking logic. We can now make a number of useful definitions.

Definition 4 *If p has the floor, then p is the speaker; otherwise, p is a listener.*

$$speaker(p,t) =_{df} has(p, \mathsf{floor}, t)$$
$$listener(p,t) =_{df} \neg speaker(p,t)$$

We have intentionally separated the notion of speaker, and the act of speaking, i.e. producing utterances. One can be the speaker in a conversation without producing vocal utterances, as in the case of dramatic pauses, or stopping to think.

We define the fluent $uttered(p,u,t)$ to mean that p uttered u in situation t. Utterances include words and gestures, such as a nod of the head or a raised eyebrow. Anyone is allowed to make an utterance at any time (so simultaneous speech is possible), and an utterance action can be performed at the same time as one of the basic single-object logic actions. Utterances have various properties that are important to turn-taking, such as volume, prosodic contour, syntactic structure, semantic structure, etc.

[8] Or humans who *really* like logic.

Domain Actions	Holder Actions (p)	Non-holder Actions (q)	Domain Actions
prevent turn loss	KEEP(p, I)	TAKE(q, I, p)	begin a turn
end a turn	DROP(p, I)	PICK-UP(q, I)	begin a turn
turn yielding signal	OFFER(p, I, q)	REQUEST(q, I, p)	turn request signal
	RETRACT-OFFER(p, I, q)	RETRACT-REQUEST(q, I, p)	
	REJECT-REQUEST(p, I, q)	REJECT-OFFER(q, I, p)	

Table 5.3: The turn-taking domain.

Definition 5 *An utterance is a speaking action performed by the speaker, and a back-channel utterance is a speaking action performed by a listener.*

UTTER(p, u)	BC-UTTER(p, u)
pre: *speaker*(p, t)	**pre:** *listener*(p, t)
effect: *uttered*(p, u, t)	**effect:** *uttered*(p, u, t)

We distinguish between speaker utterances and listener, or *back-channel*, utterances. Back-channel utterances are often confirmations (*I see*, *Yeah*, *okay*, etc.) that provide useful information, but are not meant to cause a speaker change. Back-channel utterances should usually be brief, in relation to regular utterances, since longer utterances can, sometimes unintentionally, win one the floor.

Figure 5.4 gives an example of an analysis of an ordinary piece of dialog. Figure 5.5 shows an analysis of a dialog containing many back-channel utterances.

5.3.6 Interruptions and Disruptions

A listener can try to take the floor from the speaker at any time, but if they do not follow rules for smooth turn-taking (such as those discussed in Section 5.3.9) and they take the

time	h	b	Has floor
t_1	$\begin{cases}\textsc{utter}(h, \textit{“when...”}) \\ \textsc{keep}(h, \mathsf{floor})\end{cases}$	$\emptyset$	h
t_2	$\begin{cases}\textsc{utter}(h, \textit{“when...”}) \\ \textsc{offer}(h, \mathsf{floor}, b)\end{cases}$	$\emptyset$	h
t_3	$\emptyset$	$\textsc{take}(b, \mathsf{floor}, h)$	h
t_4	$\emptyset$	$\textsc{utter}(b, \textit{“well...”})$	b
t_5	$\emptyset$	$\begin{cases}\textsc{utter}(b, \textit{“ok?”}) \\ \textsc{offer}(b, \mathsf{floor}, h)\end{cases}$	b
t_6	$\textsc{take}(h, \mathsf{floor}, b)$	$\emptyset$	b
t_7	$\textsc{utter}(h, \textit{“well...”})$	$\emptyset$	h
t_8	$\begin{cases}\textsc{utter}(h, \textit{“did...”}) \\ \textsc{offer}(h, \mathsf{floor}, b)\end{cases}$	$\emptyset$	h
t_9	$\emptyset$	$\textsc{take}(b, \mathsf{floor}, h)$	h
t_{10}	$\emptyset$	$\textsc{utter}(b, \textit{“81”})$	b
t_{11}	$\emptyset$	$\textsc{offer}(b, \mathsf{floor}, h)$	b

```
h. when did you get it? when did you put it in?
b. well the thing is it's in 6 mo certs so i got 1 6-mo. cert due next mo ok?
h. well i'm not i'm not making myself clear - did you get your lump sum in 81 or
    82
b. 81
```

Figure 5.4: Dialog from the Harry Gross transcript (Pollack et al. 1982), with no interruptions, disruptions, or back-channel utterances.

floor without requesting or being offered it, we refer to their utterance as a *turn-taking interruption.*

Definition 6 *A turn-taking interruption occurs if p takes the floor from q when no requests or offers for the floor involve p.*

$$\textit{tt-interrupt}(p, q, s) =_{df}$$

$$p \neq q \wedge \textit{speaker}(q, s) \wedge \textit{speaker}(p, \triangleright) \wedge \neg \textit{offer-pending}(q, \mathsf{X}, p, s) \wedge \neg \textit{request-pending}(p, \mathsf{X}, q, s)$$

time	h	e	Has floor
t_1	$\begin{cases}\text{UTTER}(h, \textit{“here's...”}) \\ \text{KEEP}(h, \text{floor})\end{cases}$	$\emptyset$	h
t_2	$\emptyset$	BC-UTTER(e, *“Yeah?”*)	h
t_3	$\begin{cases}\text{UTTER}(h, \textit{“you...”}) \\ \text{KEEP}(h, \text{floor})\end{cases}$	$\emptyset$	h
t_4	$\emptyset$	BC-UTTER(e, *“mhm”*)	h
t_5	$\begin{cases}\text{UTTER}(h, \textit{“your...”}) \\ \text{KEEP}(h, \text{floor})\end{cases}$	$\emptyset$	h
t_6	$\emptyset$	BC-UTTER(e, *“mhm”*)	h
t_7	$\begin{cases}\text{UTTER}(h, \textit{“your...”}) \\ \text{KEEP}(h, \text{floor})\end{cases}$	$\emptyset$	h
t_8	$\emptyset$	BC-UTTER(e, *“right”*)	h
t_9	$\begin{cases}\text{UTTER}(h, \textit{“tally...”}) \\ \text{KEEP}(h, \text{floor})\end{cases}$	$\emptyset$	h
t_{10}	$\emptyset$	BC-UTTER(e, *“mhm”*)	h

```
h. here's what you hafta do ed
e. yeah?
h. you hafta figure all of your medical expenses
e. mhm
h. your charity - ok?
e. mhm
h. your interest, your taxes and if you had any casualty losses, union dues,
    things of that nature they're also deductible
e. right
h. tally them up
e. mhm
```

Figure 5.5: Sample backchannel utterances from the Harry Gross transcript. N is not shown since it has no influence on the conversation. All of e's (the caller) utterances can be classified as backchannel since they are brief confirmations that indicate attention and, hopefully, understanding.

Figure 5.6 shows an example where the radio host Harry Gross interrupts a caller in mid-turn in order to get his name. Figure 5.7 shows an example of an interruption from

the Elhadad course advising transcripts.[9]

A *semantic interruption* is an interruption that causes the topic or focus of the conversation to change in an unexpected way. The kinds of interruptions discussed by Grosz and Sidner (1985, 1986) are semantic interruptions in our sense, and they may or may not be associated with a turn-taking interruption. For example, they give this example of a *true interruption*, where the highlighted text is part of a completely different discourse:

Dialog 5.10

> **A**: John came by and left the groceries — *Stop that you kids!* — and I put them away after he left

This is not counted as a turn-taking interruption since **A** appears to have the floor the whole time. However, N may have taken the floor momentarily if the kids in the example made a loud noise that distracted both conversants. If so, we could analyze that case as follows: **A** has the floor, N (the loud noise from the kids) takes the floor from **A**, **A** takes the floor back from N and says *Stop that you kids!*, and then continues with the interrupted discourse.

For an interruption to be warranted, the listener should believe that what it has to say is important. In general, this is a difficult issue, although there are some simple cases. For example, time-pressured utterances might sometimes be interruptions, e.g. if you see a piano about to fall on the speaker's head, it's okay to interrupt them, no matter what they are talking about.[10] What is important to one agent might not be important

[9] Available on-line via `http://www.cs.columbia.edu/~radev/u/db/acl/html/RESOURCES/CORPORA/`.

[10] Humor is often time-dependent, for instance, a pun about something the speaker has just said is often perceived as being "wittier" if it is made right after the speaker's utterance. Another example of the importance of timing, and interruptions, is given by the following joke-dialog:

Madge: Ask me what the secret of comedy is.

Trygve: What's the secr —

Madge: Timing.

time	c	h	N	Has floor
t_1	$\begin{cases}\text{UTTER}(c, \text{“}ok\ldots\text{”}) \\ \text{KEEP}(c, \text{floor})\end{cases}$	$\emptyset$	$\emptyset$	c
t_2	$\emptyset$	$\text{TAKE}(h, \text{floor}, c)$	$\emptyset$	c
t_3	$\emptyset$	$\text{UTTER}(h, \text{“}I\ldots\text{”})$	$\emptyset$	h
t_4	$\emptyset$	$\text{OFFER}(h, \text{floor}, c)$	$\emptyset$	h
t_5	$\text{TAKE}(c, \text{floor}, h)$	$\emptyset$	$\emptyset$	h
t_6	$\text{UTTER}(c, \text{“}Hank\text{”})$	$\emptyset$	$\emptyset$	c
t_7	$\text{OFFER}(c, \text{floor}, h)$	$\emptyset$	$\emptyset$	c
t_8	$\emptyset$	$\text{TAKE}(h, \text{floor}, c)$	$\emptyset$	c
t_9	$\emptyset$	$\text{UTTER}(h, \text{“}go\ldots\text{”})$	$\emptyset$	h

```
c. ok harry i'm have a problem that uh my - with today's economy my daughter
   is working -
h. i missed your name
c. hank
h. go ahead hank
```

Figure 5.6: Sample interruption from the Harry Gross transcript.

time	S	C	N	Has floor
t_1	$\begin{cases}\text{UTTER}(S, \text{“}maybe\ this\ldots\text{”}) \\ \text{KEEP}(S, \text{floor})\end{cases}$	$\text{TAKE}(C, \text{floor}, S)$	$\emptyset$	S
t_2	$\emptyset$	$\text{UTTER}(C, \text{“}What\ldots\text{”})$	$\emptyset$	C

```
S - Who are the advisors for Graduate Students here?  By
    the way, couldn't Algebra fulfill one of the electives?
    I've been thinking..a I've been thinking...
    considering to take it, I'm not sure, maybe this...

C - What number would it be?
```

Figure 5.7: Sample interruption from the advising transcript. C interrupts S by taking the floor from S before S has completely finished. Without access to the actual recordings, it is uncertain if this is really an interruption, since S might have given many of the appropriate turn-yielding signals. However, in this case, it appears that C is indeed interrupting S in order to get some relevant information.

to another, so there will always be some risk that an interruption is unappreciated, or deemed unworthy, by the speaker.

Disruptions are interruptions by N, so the above example could be classified as a disruption as well as a true interruption.

Definition 7 *A disruption is an interruption by N.*[11]

$$\textit{tt-disrupt}(s) =_{df} \textit{tt-interrupt}(N, p, s) \wedge \textit{speaker}(p, s)$$

A disruption can occur in a conversation when, for example, two people are talking, and the phone rings, causing one person to have to answer it and so disrupt the original conversation, as in Figure 5.8, taken from the Elhadad advising transcripts. Figure 5.10 gives a second example where a telephone rings, this time right in the middle of S's turn.

5.3.7 Coordinating Turn-taking with Signals

As discussed in Section 2.4, human conversationalists give signals when they give up, or are about to give up the floor, and when they want the floor. The request and offer actions are used to model *turn-yielding* and *turn-requesting* signals. A listener who wants to speak can issue a request for the floor via the REQUEST action, or instead can try to take the floor if it has been offered by the current speaker, or for some other reason (e.g. the listener has something important to say that the speaker is unaware of). Similarly, a speaker who is willing to give up the floor can offer the floor to a particular listener via the action OFFER(p, floor, q). Acknowledgments for offers and requests happen implicitly

[11] The concept of a "semantic disruption", in the sense of N causing a discourse to change its structure in an unexpected way, is possible but uncommon. One example of a semantic disruption occurred in a TV commercial about a man calling his wife from the airport, apparently saying to her "I'm ...leaving ...you.". However, it turns out that he was just using a low-quality cellular telephone, and what he actually said was disrupted by noise and had quite the opposite meaning: "I'm not leaving without you.".

time	C (Advisor)	S (Student)	N	Has floor
t_1	UTTER(C, *"Well..."*)	$\emptyset$	$\emptyset$	C
t_2	{OFFER(C, floor, S), UTTER(C, *"that's..."*)}	$\emptyset$	$\emptyset$	C
t_3	DROP(C, floor)	$\emptyset$	$\emptyset$	C
t_4	$\emptyset$	{BC-UTTER(S, *"o.k."*), PICK-UP(S, floor)}	**PICK-UP(N, floor)**	*nobody*
t_5	BC-UTTER(C, *"excuse me"*)	$\emptyset$	$\emptyset$	N
t_6	$\emptyset$	$\emptyset$	OFFER(N, floor, C)	N
t_7	TAKE(C, floor, N)	$\emptyset$	$\emptyset$	N
t_8	{OFFER(C, floor, S), UTTER(C, *"sorry"*)}	$\emptyset$	$\emptyset$	C
t_9	DROP(C, floor)	$\emptyset$	$\emptyset$	C
t_{10}	$\emptyset$	TAKE(S, floor, C)	$\emptyset$	C
t_{11}	$\emptyset$	UTTER(S, *"What..."*)	$\emptyset$	S

C: `Well, you'll be doing projects in every course...`[description of format of a semester deleted] `...and that's something for you to work out with some faculty member on something that you're interested in.`

S: `o.k.`

C: `Excuse me, (Telephone rings). telephone, For what purpose?` ...[telephone conversation deleted] `...We'll see. All right, Right. Bye. sorry.`

S: `What were you going to talk to me about - Operating Systems ...`

Figure 5.8: Sample disruption from Elhadad advising transcript (The full text is given in Figure 5.9). $\emptyset$ signifies the null action, and a boxed action indicates a competition winner. We assume after **S** says `o.k.` he wants to continue speaking, and at about the same time the phone rings. Since N has the floor when **C** utters `excuse me`, it is a back-channel utterance. Also, we assume that there is a signal in the telephone conversation to **C** that it will soon end, hence the offer by N.

by the performance of TAKE or DROP actions, or through explicit rejections, REJECT-REQUEST and REJECT-OFFER.

Here is an example of how we use turn-yielding and turn ending signals:

```
C - Well, you'll be doing projects in every course in
    homework assignments and they vary from course to
    course.  What I'm suggesting as far as the summer is
    concerned, we usually do not have any regularly
    scheduled courses but we do have project courses, and
    there you work closely with faculty members on some
    aspect of his or her research, and you'll be given some
    project to work on and that varies from hardware
    construction literature researching, writing a paper, or
    connecting software, or something similar and that's
    something for you to work out with some faculty member
    on something that you're interested in.

S - o.k.

C - Excuse me, (Telephone rings).   telephone, For
    what purpose?  O.K  Let me talk with xxx.       I've
    written it down and when I get a break, I'll talk with
    him about it.  Oh, sure.  Right.  Yes.  All right.  we
    were going to resolve these issues when we thought we
    had money.  Right.  O.K.  Let me talk with Rich and
    we'll call a meeting or we can just let him make a
    decision and go on from there, but,,, Yes.  I would
    just poll the opinions.  Right.  Or I'll speak to him
    personally.   Right,    We'll  see.   All right,  Right.
    Bye.  sorry.

S - What were you going to talk to me about - Operating
    Systems - You said you were going to talk to me about it.
```

Figure 5.9: Full advising transcript analyzed in Figure 5.8.

Dialog 5.11

(0)**A**: When are you going to England?

(1)**B**: Next year.

After utterance (0), **A** drops the floor which is meant as a turn-ending signal when done

time	S	N	Has floor
t_1	$\begin{cases}\text{UTTER}(S, u_1)\\ \text{KEEP}(S, \text{floor})\end{cases}$	$\boxed{\text{TAKE}(\text{N}, \text{floor}, S)}$	S

```
S - O.K.  One problem I do have is my Department Head
    doesn't know about these changes because he was not
    available -___

    (ringing telephone):
    yes, hi.  Yes.  Yes, I think so, right.  I asked
    somebody to do it for me.  I haven't heard.  Yes,
    what's been done.  I think Abbey was the one who was
    doing it last, I'm not sure.  O.K.  The only change
    is...the only difference with me is I'm leaving Friday
    evening rather than stay to the very end on Saturday,
    and other than that .  By the way, this
    conversation is going to be in the middle of your
    transcript.  O.K.  Right.  Bye-bye.
```

Figure 5.10: Sample disruption from the Elhadad advising transcript. A $\emptyset$ signifies the null action. Note that only S (who is *not* the same S as in Figure 5.8) and N are shown.

at the end of a grammatical utterance. Around when it is uttering "going", **A** might issue a turn-yielding signal in the form an intonation pattern or a decrease in volume. This turn-yielding signal would be an offer of the floor that remains pending until the end of the utterance, and if **B** understood what **A** was asking before the question was finished, it is permissible for **B** to try take the floor from **A** and answer the question. That case would cause a floor battle and **A** ought to relinquish the floor to **B** because **A** freely offered it. Utterance (1) is the case when **B** waits for **A** to drop the floor before speaking. **B** picks up the floor, says "next year" in response, and then drops the floor. If **B** had more to say, it might just offer the floor to **A** instead of dropping it. Now suppose that the trip to England is a surprise for **B**, and so **B** doesn't want **A** to talk, for fear of

spilling the beans. Suppose the dialog went like this:

Dialog 5.12

(0)**A**: When are you going to England?

(1)**B**: [shakes head, indicating doesn't want to talk about this]

In utterance (1), assuming **A** makes an offer, **B** rejects it by not taking the floor, instead making a gesture that indicates rejection. Technically, we treat this gesture as a kind of back-channel utterance. Figure 5.8 gives another example showing more elements of the model.

5.3.8 Resolving Floor Competitions

Given a conflicting action vector $\vec{\alpha}$, a conflict resolution procedure is necessary for determining which agent wins control of the floor. In conversation, when a conflict arises with respect to a fact, it is typically possible to resolve the conflict through a negotiation subdialog (Sidner 1994), but, when it comes to the floor, a subdialog is not an option since the very issue is who ought to speak next. We treat the conflict resolution procedure as a practical matter defined on a per-domain basis. In some situations, always letting the "higher ranking" conversant win, a kind of master/slave turn-taking relationship, might be reasonable for deciding $win(p, q, s)$. In other situations, for example an elementary school classroom, a house of parliament, or a formal debate, some turn-taking rules are explicitly given, so the specific rules can be used in those cases.

For spontaneous English conversation, Oreström (1983) lists the following factors that can influence who gets the floor:

- **volume** if you want to take control of the floor from someone, speak louder;
- **gestures** if you want the floor, make gestures for it (via back-channel utterances);

- **speaker selection** the person who currently has the floor can select who takes the floor next;
- **last speaker** when nobody has the floor, the last speaker has priority;
- **utterance content** the content of an utterance may win/lose the floor for a person.

Also, as illustrated previously, if an offer to take the floor is pending, then the speaker ought to let the offeree take it without competition. A combination of these factors can be used to decide who gets the floor in a competition. The conversant with the highest cumulative score over these features is the winner of a floor battle.

5.3.9 Standard Turn-taking Rules

Sacks et al. (1978) gives four ordered rules that describe a number of salient points about turn-taking at *transition relevance places* (TRPs) in human conversation. A TRP is a point in the speaker's speech where a floor switch is permissible. Oreström (1983) gives five features that determine a TRP:

- **prosody** completion of a tonal unit with a non-level nucleus;
- **syntax** completion of a syntactic sequence;
- **semantics** completion of a semantic sequence;
- **loudness** decrease in volume;
- **silent pause** following immediately after the end of a tonal unit.

The first three factors are referred to as a *grammatical boundary*, and according to Oreström form a "major juncture" in a discourse. The more such features that appear together, the more likely that the point in question is a TRP. Fluents for each of these

features for describing utterances can be added to the turn-taking logic in order to track them.

We summarize the turn-taking rules from Sacks et al. as follows:

Turn Taking Rules 1 (Sacks et al.)

1. *If current-speaker-selects-next technique is used by the speaker, then the selected person is obliged to take the floor;*

2. *If current-speaker-selects-next technique is not used, the first listener to self-select themselves gets the floor;*

3. *If current-speaker-selects-next technique is not used, and no one self-selects, then the current speaker keeps the floor unless he drops it.*

The current-speaker-selects-next technique refers to instances when the current speaker explicitly designates who will speak next, e.g. by saying something like *Madge, what do you think?*.

Using the turn-taking logic, it is possible to represent the turn-taking rules of Sacks et al. (1978). We assume that situations have a temporal ordering, so that it can be determined if one situation occurred before another. We use $\mathit{candidate}(p)$ to mean p is, according to these rules, the next agent expected to have the floor, and s to represent the current situation.

Turn Taking Rules 2 (Sacks et al. formalized)

1. *If the speaker p has offered the floor to a listener q, then q can take it:*

$$\mathit{offer\text{-}pending}(p, \mathsf{X}, q, s) \supset \mathit{candidate}(q)$$

2. *Otherwise, if there is a request for the floor pending, then an agent who made an earliest request can take it:*

$$\mathit{speaker}(p, s) \wedge \mathit{request\text{-}pending}(q, \mathsf{X}, p, s) \wedge \mathit{earliest}(q, s) \supset \mathit{candidate}(q)$$

where $\mathit{earliest}(q, s)$*is true when* q *is an agent who made an earliest still-pending request to the speaker.*

3. *Otherwise, the speaker keeps the floor, or can just drop the floor.*

The rules are meant to be applied in the given order, and so rule 3 is the catch-all that applies when neither of the first two do. These rules do not tell listeners or speakers *when* to offer or request the floor.

Extended Turn-taking Rules

The rules from Sacks et al. are based on analysis of English conversation and various linguistic principles. However, by exploiting just the turn-taking logic, we can extend them in a natural way. First, consider this extension (where the rules are to be applied in the order given):

Turn Taking Rules 3 (Sacks et al. extension 1)

1. N *can win any competition (depending on circumstances that cannot always be defined ahead of time);*

2. *If the speaker has offered the floor to a listener, and that listener has requested the floor, then that listener can take it:*

$$\mathit{speaker}(p, s) \wedge \mathit{offer\text{-}pending}(p, \mathsf{X}, q, s) \wedge \mathit{request\text{-}pending}(q, \mathsf{X}, p, s) \supset \mathit{candidate}(q)$$

3. *Otherwise, if the speaker has offered the floor to a listener, then that listener can take it:*

$$\textit{speaker}(p, s) \wedge \textit{offer-pending}(p, \mathsf{X}, q, s) \supset \textit{candidate}(q)$$

4. *Otherwise, if there is a request for the floor pending, then the agent who made the earliest request gets the floor*

$$\textit{speaker}(p, s) \wedge \textit{request-pending}(q, \mathsf{X}, p, s) \wedge \textit{earliest}(q, s) \supset \textit{candidate}(q)$$

5. *otherwise, the speaker keeps the floor, or can drop the floor.*

Notice that if the speaker has made an offer to an agent who has made a request, that is a better reason for giving the floor to the requester than if there is only an offer or request alone. Since the speaker has greater control over the conversation than the listener, an offer of the floor by the speaker should be a better reason for a floor switch than a request by a listener. Also, the standard rules do not rank multiple offers or requests. With these considerations in mind, we give the following rules:

Turn Taking Rules 4 (Sacks et al. extension 2)

1. *N can win any competition (depending on circumstances that cannot always be defined ahead of time);*

2. *Prefer listeners who have made a request to the speaker, and to whom the speaker has made an offer; if there is more than one such listener, break ties by ranking the listeners by the order in which they were offered the floor, which guarantees a unique earliest listener since the speaker can make at most one offer per situation;*

3. *Otherwise, prefer the listener to whom the speaker made the earliest still-pending offer;*[12]

4. *Otherwise, prefer the listener who made the earliest still-pending request;*

5. *Otherwise, the speaker can keep the floor, or just drop it.*

We do not claim this is how people handle the floor in conversation, or that this is, in general, the best protocol for human-computer interaction. Different situations sometimes use different turn-taking protocols (e.g. formal debating versus casual conversation); there is no one overall best set of turn-taking rules. Rule set 4 above is the natural and flexible from the point of view of the basic turn-taking logic, since it decides the floor taking into account both user requests and speaker offers, unlike the original Sacks et al. at rules which are asymmetric.

5.4 When to Act

A turn-taking agent works by continuously ordering all its working memory goals, and, when the circumstances are right, executing the highest-ranking goal (Section 3.3). We now explain in detail what the right circumstances are for executing a goal. In this section, we will use wm to denote working memory, and $wm[i]$ to represent the ith slot, which can contain at most one goal. After working memory has been properly ordered, the highest ranking goal will be in $wm[0]$.

The following will be from the perspective of a single agent taking a turn in a conversation. This is in contrast to the turn-taking logic of Section 5.3, which provides an omniscient third-party perspective on the interaction. The turn-taking logic, as part of

[12]Breaking ties through the earliest still-pending offer is based on the second rule from the Sacks et al. turn-taking rules. Empirical study of discourse, similar to the studies carried out by Sacks et al., would be necessary to determine the validity of this rule.

the general three-step turn-taking model (Chapter 3) provides the basic concepts and features an agent-oriented turn-taking algorithm can consider.

5.4.1 Transition Relevance Places

As mentioned in Section 5.3.9, a transition relevance place, or TRP, is point in a conversation where a floor shift can happen smoothly. In order to know when best to speak, conversants must be aware of TRPs, both how to recognize them and when to signal them to the other conversant; the speaker-shifts specified by the rules in Section 5.3.9 are meant to take place at TRPs. In the model presented in this section, TRPs will be represented by conversational signals. A speaker can issue a *turn-yielding* or *turn-ending* signal, while a listener can issue a *turn-request* signal, or simply attempt to take the floor. A speaker should issue a turn-ending signal when it gives up the floor, and it should issue a turn-yielding signal before it issues a turn-ending signal, to allow the listener to prepare for an upcoming floor transition. The frequency and location of turn-yielding signals depends on many factors, such as the general conversational traits of the individual speaker, their personal goals, their relationship with the other conversant, the topic being discussed, etc. Since the agents we want to design with this theory are mainly computer assistants collaborating with a human user, it makes sense to allow more opportunities for the user to speak rather than fewer. In practice, even in a mixed-initiative system, the user is "in charge" because the problem being solved is *their* problem, and people typically have other concerns unrelated to the collaborative task that can nonetheless influence it. For example, a person might be rushing to make an appointment, or be sleepy or hungry, and so this could influence their behaviour with respect to the problem domain.

As represented in Figure 5.11, the conversational signals are derived from the turn-taking logic of Section 5.3.5. Symbolically, a turn-yielding signal will be treated like a

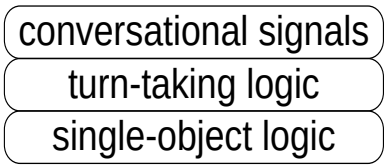


Figure 5.11: Layers of logical representation.

special kind of punctuation symbol, and it will be taken to mean that the speaker will soon give up the floor. In practice, one cannot always expect to see/hear an explicit turn-yielding signal the way one sees a comma in a written sentence or hears a pause in a spoken one. Instead, in human-human conversation, turn-yielding signals are usually signaled and recognized implicitly by the co-occurrence of particular grammatical and vocal features, such as a decrease in speaking volume, change in pitch, and a grammatical boundary (Section 2.4). Symbolically, a turn-ending signal is also treated like a special kind of punctuation, somewhat analogous to sentence-ending punctuation.[13] A turn-ending signal is taken as a sign that the agent has given up the floor, and so after that signal the agent no longer has the floor, unless it explicitly picks it up again. Again, speakers do not always make turn-ending signals explicit, although they are somewhat easier to identify than turn-yielding signals because they typically correspond to the ends of sentences (and are then followed by silence until someone picks up the floor). Turn-yielding and turn-ending signals typically appear together, with the turn-yielding signal coming before and indicating an upcoming turn-ending signal. A turn-yielding signal is not required, but it is highly desirable, since it helps prepare the listener for their next turn. Without turn-yielding signals, turn-ending signals would appear "out of the blue", and turn-taking would not happen as smoothly as it does for most people.

[13] More specifically, a turn-ending signal is like a ".", "!", or "?" at the end of a sentence. There is no sentence-punctuation analogue for a turn-yielding signal, i.e. a punctuation symbol that means "the end of the sentence is coming up soon".

5.4.2 Transition Actions

We will suppose that an agent uses at least four pieces of information in deciding what action to take with respect to the floor. Four flags describe the state of the agent relevant to taking a turn; a flag set to 1 means true, and 0 means false. We will also use *TTgoal* to mean an external goal (Section 3.4) that requires possession of the floor for the agent to be able to execute it. The four flags are:

has floor (HF) set exactly when the agent has the floor;

ordering done (OD) set exactly when ordering of working memory is finished; **OD**=1 does not necessarily mean that the goals are in their optimal ordering, just that the algorithm ordering them will not do any better;

$wm[0]$ **is a TTgoal (TT)** set exactly when $wm[0]$ is a TTgoal;

conversant signal (CS) the five possible values for the conversant signal are:

- **none** no signal;
- **drop floor** the speaker has dropped the floor, i.e. given a turn-ending signal;
- **offer floor** the speaker has made an offer of the floor;
- **request floor** the listener has requested the floor;
- **take floor** the listener has attempted a floor-taking action.

Along with the floor-related actions given in Section 5.3.5, we recognize the following three actions:

continue keep doing what you are doing (e.g. remain the speaker or listener);

stall stall for time, by inserting a stock-phrase that indicates the reason for the stall (Section 5.4.3);

execute attempt the best TTgoal, which, when **TT**=1, is $wm[0]$; otherwise, the TTgoal with the smallest index can be executed.

For simplicity, we will assume there are no coordination problems, so that if an agent gives a signal or performs an action, the other agent will correctly recognize it. Of course, in practice, this assumption is too strong, but it is a useful initial assumption, and weakening it is an interesting problem for future research.

Table 5.4 summarizes the actions that an agent should take in a conversation, and will be further discussed in Section 5.4.4.

5.4.3 Stalling

A characteristic of human-human conversation is the presence of stalling phrases that often do little more than buy extra time for the speaker to prepare their main utterance. Stalling phrases can be short, such as *hmmm*, *uh*, *well*, or longer canned phrases like *let me see*, or *that's a good question.*[14] Individual speakers can have their own characteristic use of stalls. Of course, we do not want to create systems that make human-like stalls (or mistakes) if they can be avoided, but the value of adding stalls is that they can provide useful feedback to the user about why the system is not responding immediately.

As shown in Figure 5.12, stalls are divided into two types, those that are self-initiated and are used to buy thinking time, and those that are meant to rebuff a turn-taking request from the other conversant. A buy-time stall could just be a pause, where the agent simply waits until it has something reasonable to say.[15] A filled buy-time stall comes in two types, depending on the state of the agent's processing. If the agent has not yet completed its processing, then it can issue a thinking-stall, to indicate it is not

[14] We do not mean to say that such phrases are only or always stalls for time, just that they are often used as such.

[15] Such a pause should be distinguished from other kinds of pauses, such as a dramatic pause.

HF=has floor?, **OD**=ordering done?, **TT**=next goal a TTgoal?, **CS**=conversant signal?

	HF	OD	TT	CS	Action
		0	0	none	CONTINUE (listening)
		0	0	drop floor	CONTINUE (don't take floor)
		0	0	offer floor	REJECT OFFER
		0	1	none	CONTINUE (listening)
		0	1	drop floor	CONTINUE (don't take floor)
		0	1	offer floor	REJECT OFFER
Listener					
		1	0	none	CONTINUE (listening)
		1	0	drop floor	TAKE FLOOR and ELICIT-STALL
		1	0	offer floor	REJECT OFFER
		1	1	none	REQUEST FLOOR (importance)
		1	1	drop floor	TAKE FLOOR and EXECUTE
		1	1	offer floor	TAKE FLOOR and EXECUTE
		0	0	none	THINKING-STALL
		0	0	request floor	REJECT REQUEST
		0	0	take floor	KEEP FLOOR
		0	1	none	STALL
		0	1	request floor	REJECT REQUEST
		0	1	take floor	KEEP FLOOR
Speaker					
		1	0	none	OFFER FLOOR and ELICIT-STALL
		1	0	request floor	DROP FLOOR
		1	0	take floor	CONTINUE (release floor)
		1	1	none	EXECUTE
		1	1	request floor	REJECT REQUEST and EXECUTE
		1	1	take floor	KEEP FLOOR and EXECUTE

Table 5.4: Action table. Rows above line are listener actions (**HF**=0), and rows below the line are speaker actions (**HF**=1).

responding because it is still thinking; stalls in this category would include utterances such as *let me think*, *just a moment*, or even *hmmmm*. If the agent has finished processing but no turn-taking goal has been ranked high enough, the agent can try to elicit behaviour from the other conversant that will cause more goals in the agent to be triggered. Phrases to do this would be tailored towards the current situation, and would generally include utterances such as *tell me more*, *I'm not sure how to respond*, *I need more information*, etc. It is possible for a thinking stall to be followed by an elicit stall, and stalls need not be linguistic, but could, for example, be realized through hand gestures, facial expressions, or body language.

Rebuffing stalls can be used in conjunction with rejection actions, when a speaker is rejecting a floor request or when a listener is rejecting a floor offer. For example, a speaker might rebuff a listener request as follows:

Dialog 5.13

Madge: [Referring to Buckingham palace] I wonder who lives in that house?

Trygve: [Indicates he wants to answer this question, but Madge does not give up the floor]

Madge: That was a *rhetorical question.* Everyone knows it's the Spicegirls.

Here is an example of a rebuff-stall used by a listener to reject a floor offer:

Dialog 5.14

Trygve: My friend here might know the York transit system ... but maybe not.

Madge: [shakes head 'no' after first part of utterance; she is rejecting the speaker's offer of the floor]

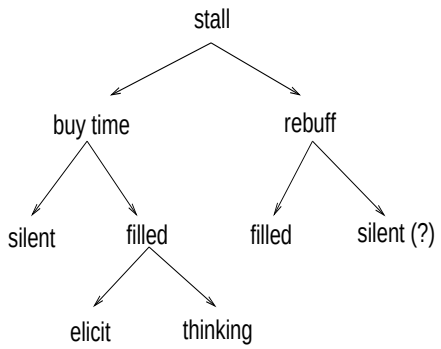


Figure 5.12: Types of stalls.

In the case of a stall from a listener, the utterance should be brief to make sure it is a back-channel utterance, otherwise the listener could end up with the floor, which they explicitly do not want.

Another case where stalling phrases could come in handy is when a speaker asks a listener a question, offers the user the floor, but keeps the floor while the listener is thinking. In this case, the listener might want to use back-channel utterances to indicate he is considering the question. For example:

Dialog 5.15

Madge: Where did you put the shoes? Any idea? Any idea at all? Even the faintest memory? [...]

Trygve: [back-channel utterances made after Madge's first question] hmmm ...maybe ...I think ...well ...

The rules of Table 5.4 do not cover this case because it requires the listener recognizing that they have been asked a question (hence made an offer of the floor), and so we leave this and similar examples for future research.

5.4.4 Examples

In the following two sections, we discuss and give examples for each row of the rule-table in Table 5.4, divided into listener and speaker cases.

Listener Examples

OD=0, TT=0, CS=none $\longrightarrow$ continue (listening)

OD=0, TT=0, CS=drop floor $\longrightarrow$ continue (don't take floor)

OD=0, TT=0, CS=offer floor $\longrightarrow$ reject offer

In these three situations, the agent is a listener who has not yet finished ordering its goals, so it is not ready to say anything. Hence, in the first case, where the speaker is making no floor-related request to the agent, the agent should do nothing and continue listening. If the speaker drops the floor, then the agent should not pick it up until it has something to say; otherwise it will immediately need to make a thinking-stall. Note that once the agent has finished ordering its goals, it will then pick up the floor if it is still available, and execute the first TTgoal or make an elicit-stall. In practice, it can be hard to distinguish between an agent who has refused to pick up the floor and an agent who has picked up the floor but has not said anything. In these rules, the difference is clear because an agent who picks up the floor is obligated to make a stalling utterance, to indicate that they have the floor and are trying to think of something to say for their turn. For example:

Dialog 5.16

Madge: Do you know where we are? [drops floor]

Trygve: [picks up floor] Ummm . . . ahhhh . . . let me see . . .

If the agent does not pick up a just-dropped floor, then it is not required to say anything; hence, an "awkward pause" can result. For example:

Dialog 5.17

Madge: Do you know where we are? [drops floor]

Trygve: [Does not pick up floor, and says nothing. An awkward pause results.]

Since the rules of Table 5.4 are biased towards *not* pausing when an agent has the floor, an agent who is aware of these rules can interpret a long pause in a conversation as an indication that no one has the floor.

In the third of the above rules, an agent can reject an offer of the floor using a brief back-channel utterance, i.e. some brief indication to the listener that they do not want the floor. For example:

Dialog 5.18

Madge: Do you want to pick a movie, or ... [floor offer]

Trygve: [rejects offer by shaking head "no"]

Madge: ... or should we see a play?

In some cases, the listener need not explicitly reject a floor offer, for example if the speaker is holding onto the floor and issuing a series of stall-like phrases. For example:

Dialog 5.19

Madge: Let's see now ... [stall/floor offer] I'm not sure ... [stall/floor offer] if we should turn left or right. Okay, let's try left.

OD=0, TT=1, CS=none $\longrightarrow$ continue (listening)

OD=0, TT=1, CS=drop floor $\longrightarrow$ continue (don't take floor)

OD=0, TT=1, CS=offer floor $\longrightarrow$ reject offer

The above three rules are very similar to the previous three, except that there is currently a TTgoal in $wm[0]$. Such an occurrence might just be luck due to the operation of the ordering algorithm, so it is generally unwise to interpret this situation as being substantially different than the one where the previous three rules applied. The local search algorithms given in Section 4.4 work such that $wm[0]$ is settled upon before locations later in wm are chosen, so those algorithms can set **TT**=1 and promise that **TT** will not become unset unless new goals appear in wm.

OD=1, TT=0, CS=none $\longrightarrow$ continue (listening)

OD=1, TT=0, CS=drop floor $\longrightarrow$ take floor and elicit-stall

OD=1, TT=0, CS=offer floor $\longrightarrow$ reject offer

Here, the agent has finished thinking, but what it has chosen to do next is not a TTgoal. In general, an agent, such as a human, will have many things on their mind, and while usually only goals relevant to the current discourse are being considered, it is possible for other things to arise as being more important. One possibility that this situation suggests is an end to the conversation, especially if there are no other relevant TTgoals in working memory. We assume that an agent will follow a more or less scripted routine for ending a conversation, the details depending on the type of the interaction; e.g. a leisurely conversation between friends will likely end differently than a rushed conversation between co-workers, or a conversation between people who do not know each other well.

In the second rule, where the speaker drops the floor, the agent ought take the floor to avoid an awkward pause. For example:

Dialog 5.20

Trygve: That's it, we're lost. [drops floor]

Madge: [picks up floor] Umm ...let me think ...do we recognize anything? ...are there any street names? ...

Here Madge takes the floor that Trygve has dropped, and begins to utter stalls and elicitation phrases in an effort to generate a reasonable next utterance. The stalls also come with floor-offers, so that Trygve can take back the floor without interrupting Madge.

The third rule requires the agent to reject an offer from the speaker if $wm[0]$ does not contain a TTgoal. For example:

Dialog 5.21

Trygve: We could talk about fan fiction [floor offer] ...

Madge: Uh-uh! [rejects offer with a back-channel-utterance, since she has nothing to say about fan fiction]

As in Section 5.4.3, Madge's utterance can be described as an offer rebuff.

OD=1, TT=1, CS=none $\longrightarrow$ request floor (importance)

OD=1, TT=1, CS=drop floor $\longrightarrow$ take floor and execute

OD=1, TT=1, CS=offer floor $\longrightarrow$ take floor and execute

When a listening-agent has decided what it wants to do/say, the above rules determine that it ought to go ahead and do it, although within the bounds of smooth turn-taking. In the first rule, where the speaker is holding onto the floor with no offers, the agent ought to request the floor; if, before getting the floor, another goal is added to the agent's

working memory causing the ordering to become different, then the agent can retract this request. Also, this is the case where the agent can consider possibly interrupting the speaker and engaging in a floor-battle, as discussed in Section 5.3.6. In the last two rules, the agent takes the floor and attempts the goal in $wm[0]$.

Speaker Examples

The following are dialog examples for the bottom 12 rows of Table 5.4, corresponding to the actions the agent can perform as speaker (i.e. **HF**=1).

OD=0, TT=0, CS=none $\longrightarrow$ thinking-stall

OD=0, TT=0, CS=request floor $\longrightarrow$ reject request

OD=0, TT=0, CS=take floor $\longrightarrow$ keep floor

These three rules cover the case when a speaker is not yet prepared to speak. When there is no signal from the listener, then the speaker ought to keep the floor and make thinking-stalls until it has finished ordering its goals in wm. For example:

Dialog 5.22

Madge: [not yet finished ordering goals] let me think ... which way, which way ... [finishes ordering goals] Okay, Pizza Hut is to the left.

If the listener requests the floor or attempts to take it, then the second rule directs the speaker to try to keep the floor. For example:

Dialog 5.23

Madge: [not yet finished ordering goals] let me think ... which way ...

Trygve: I am —

Madge: Shhhh! I'm trying to think ... which way ... [finishes ordering goals] Okay, Pizza Hut is to the left.

OD=0, TT=1, CS=none $\longrightarrow$ stall

OD=0, TT=1, CS=request floor $\longrightarrow$ reject request

OD=0, TT=1, CS=take floor $\longrightarrow$ keep floor

As in the corresponding rules for a listener, the above three rules handle the case when $wm[0]$ is a TTgoal but goal ordering is not completed. As mentioned, the local search algorithms of Section 4.4 guarantee that if they set **TT**=1, then **TT** will not become unset unless new goals are added to wm.

OD=1, TT=0, CS=none $\longrightarrow$ offer floor and elicit-stall

OD=1, TT=0, CS=request floor $\longrightarrow$ drop floor

OD=1, TT=0, CS=take floor $\longrightarrow$ continue (release floor)

When the speaker has finished ordering its goals, but $wm[0]$ does not contain a TTgoal, then the speaker either needs to give up the floor, or make elicit-stalls in hoped of getting the listener to request the floor, or say something that will cause another goal to be added to the speaker's working memory. The first rule directs the speaker to stall in manner that could elicit either a request for the floor from the user, or new information that could add goals to the speakers working memory. For example:

Dialog 5.24

Trygve: [finished ordering goals] I'm out of ideas here ... any ideas? [floor offer] ... any thoughts? [floor offer] ...

Theoretically, an agent could continue this forever, if the other agent does nothing. However, we will assume that eventually some other process in the sub-system will post a goal to working memory indicating that the conversation is stuck, or that the other agent is not participating in it any more. The last two rules take advantage of the desire from the listener to take the floor, and allow this to happen without incident: a request from the listener will cause the speaker to drop the floor (so the listener can pick it up), and if the listener attempts to take the floor from the speaker, then it will let the listener win the floor-battle. In the latter case, the speaker may have offered the listener the floor via elicit-stalls, so this is in line with the turn-taking rules of Section 5.3.9.

This case is also relevant to ending a conversation. If two people have completed their task, then there is nothing left to say relevant to this conversation, and so it should be brought to a close. We assume that a sub-system related to task-performance will post a goal indicating the task is finished, and so this circumstance then becomes a special case, and instead of stalling, the speaker should enter a conversation-closing script. For example:

Dialog 5.25

Madge: [Realizing they have escaped from Soho] Good, we made it. [Closing script] Thanks for all your help.

Trygve: No problem.

This closes off the interaction related to being lost in Soho, which could just be one subdialog of a much larger dialog.

OD=1, TT=1, CS=none $\longrightarrow$ execute

OD=1, TT=1, CS=request floor $\longrightarrow$ reject request and execute

OD=1, TT=1, CS=take floor $\longrightarrow$ keep floor and execute

A speaker will always try to make an utterance when ordering is done and $wm[0]$ has a TTgoal. Since the listener has not yet heard the speaker's utterance, the agent rejects any listener request, or tries to prevent the listener from taking the floor.

5.5 Summary

This section has presented both a general, third-party model of turn-taking (Section 5.3), and an agent-oriented algorithm based on this logic that explains what actions an agent should take depending on (at least) the processing state of the agent, and the signals from the other conversant (Section 5.4). In Chapter 6, we will present, and analyze in terms of turn-taking, two course-advising systems that illustrate how the turn-taking model can be applied to actual systems.

Chapter 6

An Application to Interface Design

Remember: Always let your conscience be your guide. *The Blue Fairy, Pinocchio*

6.1 Introduction

This chapter shows how the turn-taking model discussed in the previous chapters can be applied to the analysis of interactive systems. We will present a general turn-taking checklist, and then apply it to the analysis of two different course-advising systems:

- *SG*, a command-line system where the user types in pseudo-natural language queries and commands. It uses the constraint-based goal-ordering of Chapter 4;
- SAM, a graphical user interface that lets a user construct a schedule of courses by directly selecting courses on-screen with a mouse.

Analyzing these systems in terms of turn-taking provides a number of interesting and useful insights that could be applied in the development of future versions of these pro-

grams.

6.2 A Developer's Guide to Turn-taking

Figure 6.2 gives a list of questions that a system designer should ask when analyzing a program as a turn-taking system. The *SG* (Section 6.4) and SAM (Section 6.5) advising systems were developed essentially independently of the turn-taking model, and we analyze them in terms of it using the questions from Figure 6.2 as a guide. In both cases, this analysis clarifies the dialog-perspective of the system, and suggests a number of ways the systems could potentially be redesigned and improved.

Landauer (1995) argues that the gold standard for usability is user testing. It is difficult to disagree with this point of view, since a system that users do not like using is a failure, no matter how elegant, intelligent, or internally consistent it is. Thus, without user testing, we cannot directly argue that analyzing systems using the checklist of Figure 6.2 will result in better, or more usable, interactive systems. However, if a developer treats a particular human-computer interaction as a kind of dialog, then it makes sense to take the dialog metaphor seriously, and answering the questions of Figure 6.2 provides a systematic and theoretically defensible way of determining how well the dialog metaphor actually fits. Furthermore, as discussed in Section 6.4 and Section 6.5, the analysis can lead to a better understanding of the system for the developer, potentially suggesting extensions and improvements. Figure 6.1 gives a list of ten established usability heuristics that are meant to be applied to the analysis of a system, and the result is a list of questions that can help the developer extend, fix, or better understand their system. The questions of Figure 6.2 are meant to play a similarly suggestive role.

Figure 6.2 does not provide a detailed set of questions for the reasoning step of the turn-taking model. This is because the checklist is intended to assist a system designer

1. Use simple and natural dialogue. Tell only what is necessary, and tell it in a logical order. Ask only what users can answer.
2. Speak the users' language. Use words and concepts familiar to them in their work, not jargon about the computer's innards.
3. Minimize the users' memory load by providing needed information when it's needed.
4. Be consistent in terminology and required actions.
5. Keep the user informed about what the system is doing.
6. Provide clearly marked exits so users can escape from unintended situations.
7. Provide shortcuts for frequent actions and advanced users.
8. Give good, clear, specific, and constructive error messages in plain language, not beeps or codes.
9. Wherever possible, prevent errors from occurring by keeping choices and actions simple and easy.
10. Provide clear, concise, complete online help, instructions, and documentation. Orient them to user tasks.

Figure 6.1: Interface evaluation heuristics, from Nielsen (1993), as summarized by Landauer (1995, p.283–4). Each heuristic suggests a number of questions that the developer should ask of their system. A number of these heuristics are encapsulated in the turn-taking question template of Figure 6.2. For example: heuristic 4 is followed since the turn-taking model provides a unified and systematic description of interaction from the point of view of turn-taking; heuristic 5 is partially addressed by the use of turn-yielding signals; and some situations covered by heuristic 6 can be counted as user-initiated interruptions.

in analyzing an existing system, that will already use a particular reasoning system. In Chapter 4, it is argued that the next-action problem solved by local search is a good reasoning process to use if one is designing a turn-taking system "from scratch".

Motivation What motivates the system to take a turn? Is the processing associated with this step resource-sensitive? Can it tolerate interruptions or partial solutions?

Reasoning Is the processing resource-sensitive? Can it tolerate interruptions or partial solutions?

Turn-taking

- **utterances** Are the user and system utterances of the same type? Are there important differences?
  - **User** What is an utterance? What kinds of utterances are possible? When can they occur?
  - **System** What is an utterance? What kinds of utterances are possible? When can they occur?
- **back-channel utterances** How many, and what kind, of channels of communication exist?
  - **User** What is a back-channel utterance? What kinds of back-channel utterances are possible? When can they occur? What do they signify?
  - **System** What is a back-channel utterance? What kinds of back-channel utterances are possible? When can they occur? What do they signify?
- **turn-taking signals**
  - **User**
    - **turn-ending signal** What are the turn-ending signals? How are they recognized by the system?
    - **turn-yielding signal** What are the turn-yielding signals? How are they recognized? What do they indicate to the system? Are they explicit, implicit, or a mixture of both?
    - **interruptions** When can a user interrupt the system? What are the relevant TRPs (transition relevance places)? Why would the user interrupt?
  - **System**
    - **turn-ending signal** What are the turn-ending signals? How are they recognized by the user?
    - **turn-yielding signal** What are the turn-yielding signals? How are the recognized? What do they indicate to the user? Are they explicit, implicit, or a mixture of both?
    - **interruptions** When can the system interrupt the user without irritating them? What are the relevant TRPs (transition relevance places)? Why would the system interrupt?
  - **disruptions** What kind of disruptions can occur? Can they influence the course of the interaction?

Figure 6.2: Questions for system designers to ask. This provides a set of questions that a system designer can use to analyze a given interactive system in terms of turn-taking. The answers to these questions provide a potentially new and systematic perspective on the system, and can lead to suggestions that improve its design and behaviour.

6.3 Course Advising

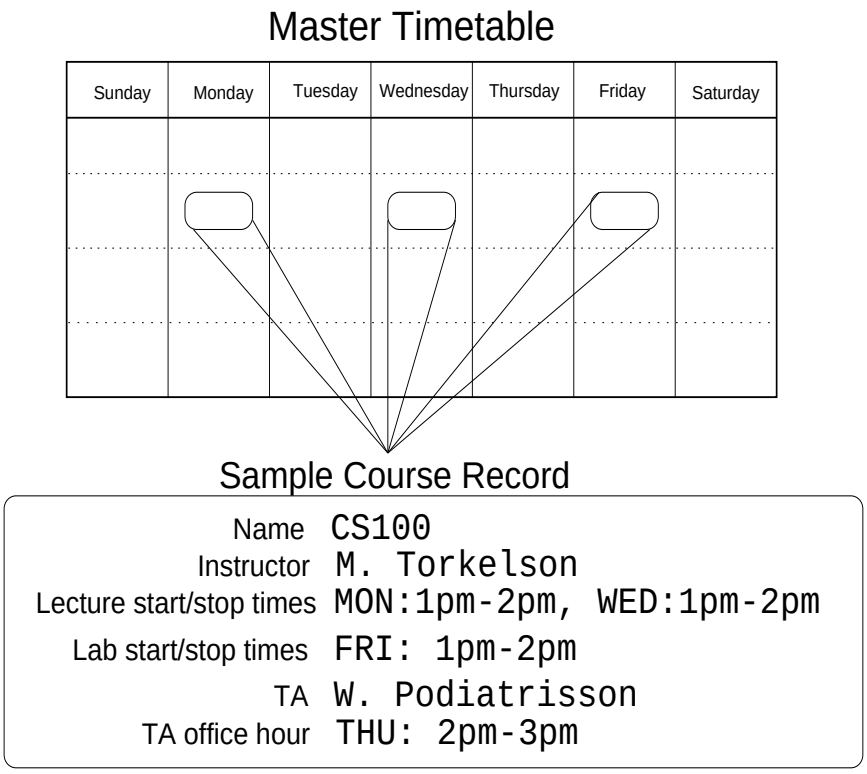
Course advising is a common domain of application for discourse systems (e.g. Carberry 1990). A typical course advising session occurs between a student and an expert advisor. We assume that the timetable of courses offered by the school has already been created, and contains information about when courses are offered, who teaches them, what they are about, what the pre/anti/corequisites are, etc. (Figure 6.3). Students must create a schedule of five or so courses, such that the following hard constraints are satisfied:[1]

- no two courses occur at the same time (i.e. none of the times for the labs, lectures, or tutorials overlap);
- the student has satisfied all the prerequisites for all the courses;
- the student has taken none of the antirequisites of the courses (a course is an antirequisite of itself, so a student cannot take the same course again);
- the student is taking any necessary corequisites.

Once these hard constraints are satisfied, then student preferences can be taken into account. For example, a student might prefer classes taught by a certain teacher, or classes offered at a certain time. Such preferences are soft constraints because it is more important to satisfy all the above hard constraints than to fail to satisfy any of these preferences.

In practice, we must put limits on the kinds of situations a course advising system can handle, but the ultimate goal is to create a system that could substitute for a real human advisor. Thus essentially any of the problems, questions, or concerns that arise in

[1] Course A is a *prerequisite* of course B if A must be completed before B can be taken; A is an *antirequisite* of B if having taken A precludes the possibility of taking B; and A is a *corequisite* of B if to take B course A must be completed during the same semester as B.



Sample Student Schedule

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
X	X	ENG 200	ENG 200	X	WRI 100	X
X	MATH 234	STAT 220	MATH 234	STAT 220	MATH 234	X
X	PHIL 330	X	PHIL 330	PHIL 330	X	X
X	MATH 251	WRI 100	ENG 200	MATH 251	MATH 251	X

Figure 6.3: Timetables and schedules. A block diagram is given in Figure 6.4. The Master Timetable contains all the information about courses for a particular school semester. The position(s) of a course in the schedule graphically show when it is offered, and for how long. In the Master Timetable, more than one course could appear at the same position. Creating the Master Timetable is a hard scheduling problem known as a *timetabling* problem (Burke, Jackson, Kingston, and Weare 1997), and a school will need to make at least one such timetable per term; teacher preferences can be used as soft constraints here. The Student Schedule, or schedule for short, is based on the Master Timetable. As discussed in the text, such a schedule is created by selecting courses from the Master Timetable that break no hard "requisite" constraints, and satisfy as many of the student preferences as possible. Note that there will need to be one Student Schedule per student each semester, as compared to a single Master Schedule per semester.

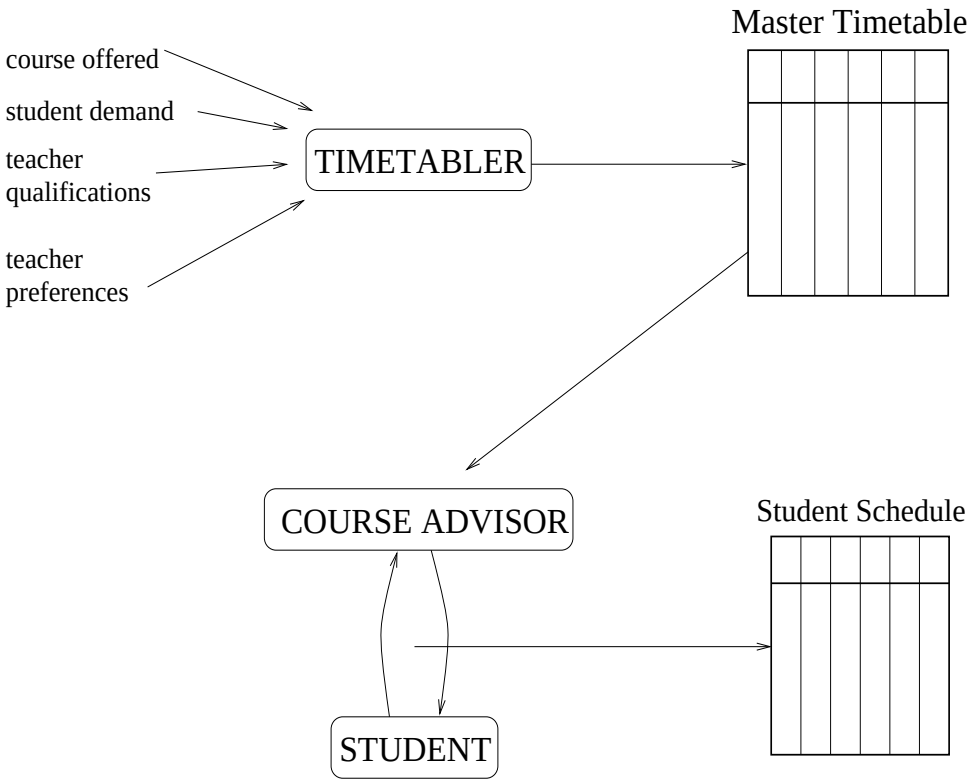


Figure 6.4: Block diagram for timetabling and course advising. The master timetable and student schedule are structured as shown in Figure 6.3. The student schedule is the product of the interactive collaboration between the course advising system and the student.

real course advising situations could be added to this application. Also, in the real world of course advising, there can be many exceptions to the rules. For example, the point of prerequisites is to ensure that students have the relevant background, but sometimes students can have that background without having taken the listed courses, e.g. they have transferred from another school, or have gained the needed experience in a job. Thus, there can be cases where a student has taken course, but none of its supposed prerequisites; this could present problems when deciding what other courses are necessary.

Another complication, at least at universities, is that rules are sometimes relaxed for graduate students who take undergraduate courses. For instance, a Ph.D. comprehensive committee might require that you complete two entire hardware courses within a year, say CS450 and its prerequisite CS351. However, CS450 must be taken first since it is offered only once a year, and at the same time as CS351. In other words, the prerequisite schedule constraint has been relaxed in favour of other constraints that are judged to be more important.

The ideal course advising system must be able to deal these and similar exceptions. In general, this is a difficult problem, but one potentially practical solution would be to apply case-based reasoning (Kolodner 1993). A database of exceptional cases could be stored, and these cases can be retrieved by the system and be compared to the current situation. If the current situation is similar enough to one of the stored exceptional cases, actions similar to those taken in the stored case can be taken in the current situation. This case-based approach makes sense with exceptional cases since they are likely rarer than ordinary cases, and ordinary cases can be handled via rules as done in *SG*.

6.4 *SG*: An Implementation of a Course Advising System

SG is a simple course advising system implemented in C++ and PROLOG.[2] It is not meant to be a production-quality system, but was intentionally designed as a prototype for exploring some of the ideas developed in this thesis. The distinction between domain-specific and domain-independent aspects of this system is clean, so it should be straightforward to apply *SG* to different advising domains (e.g. cooking assistant, waiter/cashier, ...). Figure 6.5 gives a transcript of a simple dialog between *SG* and a student.

The input to *SG* is an English expression, such as *Can I take CS130?*. The phrase is tokenized, and rules are used to determine what goals to place into working memory. Special goals for recognizing the first word, unknown words, and questions about courses and teachers are posted here. The goals are selected and ordered according to the next-action problem, and solved using the local search algorithms (Section 4.4). When the ordering has finished, the goal at the head of working memory represents the one that should be attempted next, and the remaining goals are listed in the order in which they should be achieved. The initial goal is executed, resulting in some kind of interaction with the user, i.e. displaying a message or getting information from the user. Goals may be added to working memory or deleted from it, and so the ordering algorithm is called again in these cases.

To avoid full-blown parsing of natural language input, the first word of the user's input is used to decide the type of the input. *SG* currently handles these questions:

- **can** I take course C?
- **who** teaches course C?

[2] *SG* may — or may not — be an abbreviation for *Spicegirl*. I'd rather not say.

```
User> can I take cs100?
  SG> You have already taken CS100, so no, you cannot take it again this term.
User> can I take cs212?
  SG> Yes, you can take CS212, since it is offered this term.
User> who teaches cs212?
  SG> CS212 is taught by [KIERSTEAD]
User> take cs212
  SG> Okay, adding CS212 to your schedule.
      Courses selected so far: [course(CS212)]
User> take cs100
  SG> You have previously taken CS100, so you cannot take it again.
      Courses selected so far: [course(CS212)]
User> drop cs100
  SG> You cannot drop CS100 because you have previously taken it, so it cannot be
      on your current schedule.
      Courses selected so far: [course(CS212)]
User> drop cs212
  SG> Okay, dropping CS212 from your schedule.
      Courses selected so far: []
```

Figure 6.5: Sample sessions with *SG*. In this example, it is assumed that the student has previously taken CS100, CS102, and CS112. *SG* will not allow the student to construct a schedule that breaks any hard constraints, so a student cannot take two courses that occur at the same time, or take courses for which she does not have the necessary prerequisites, etc.

- **does** teacher T teach course C?
- **what** courses does teacher T teach?

It also handles two commands:

- **take** course C
- **drop** course C

If the first word of the user's input is not one of the five bold-faced words, then *SG* automatically presents the user with a menu of legal first words to choose from. In general, if the user does not supply a needed piece of information, *SG* asks the user to make a selection from a menu of all possibilities.

SG relies on an understanding technique similar to that used by ELIZA (Section 6.4.3). For example, when answering the question *Can I take cs100?*, *SG* only cares about the word **can**, and the course **cs100**; the other words are ignored, but in an interesting way. *SG* tokenizes every word the user types, and if it sees a word that it is not familiar with, then it posts an Unknown_Word goal that, if executed, will ask the user to select from the list of words that *SG* does know. However, this never happens in the current *SG* system, since achievement of an Unknown_Word goal is of lower priority than achievement of all other goals (see Figure 6.10 and Figure 6.11). This means that other goals will be attempted before an Unknown_Word goal, and these other goals are designed such that that they will step the user through a series of questions if insufficient information is provided. As shown in Figure 6.11, Unknown_Word goals are always deleted once a goal that dominates them is achieved. As an example, Figure 6.6 shows a dialog with *SG* where a misspelled word is recognized (i.e. an Unknown_Word goal is actually posted), but it is removed without ever being attempted since the Can_Take_Course goal achieved first deletes it. Again, this technique is just an implementation convenience, and can be

```
User> can i tak cs130
  SG> Yes, you can take CS130, since it is offered this term.
```

Figure 6.6: Sample spelling mistake that is ignored by *SG*. The token "tak" should be the word "take", but *SG* realizes it does not need to find out what "tak" really means since it will have no bearing on the interaction.

replaced with more sophisticated parsing and semantic analysis; however, the technique may be of some interest itself, since it allows any circumstance that can be recognized to be posted as a goal and then explicitly reasoned with.

Figure 6.7 shows the hierarchy of goals used by *SG*. Figure 6.10 shows the penalty matrix used to order goals in *SG* (described below). Note that only the leaf, or *concrete*, classes are shown in the penalty matrix; the abstract interior classes are never instantiated because they are only there for implementation reasons (i.e. they contain shared code).

PROLOG was used for the course database because it is a simple and implementation-efficient language for databases like *SG*'s. Arbitrary queries and transformations can be made on the data, and it is straightforward to call specific PROLOG queries from C++. Technically, at the beginning of each run, *SG* gathers all the relevant data by making the appropriate general queries on the PROLOG database, storing the results in memory as vectors. This is all done automatically in a wrapper-object, so the client code in *SG* is immune to (and does not care about) the details of how the database is accessed. This method conveniently provides both the flexibility and expressiveness of PROLOG, and the modularization of object-oriented programming. If, for example, one wanted to replace PROLOG with a relational database, then all that is necessary is to change the implementation of the one database object.

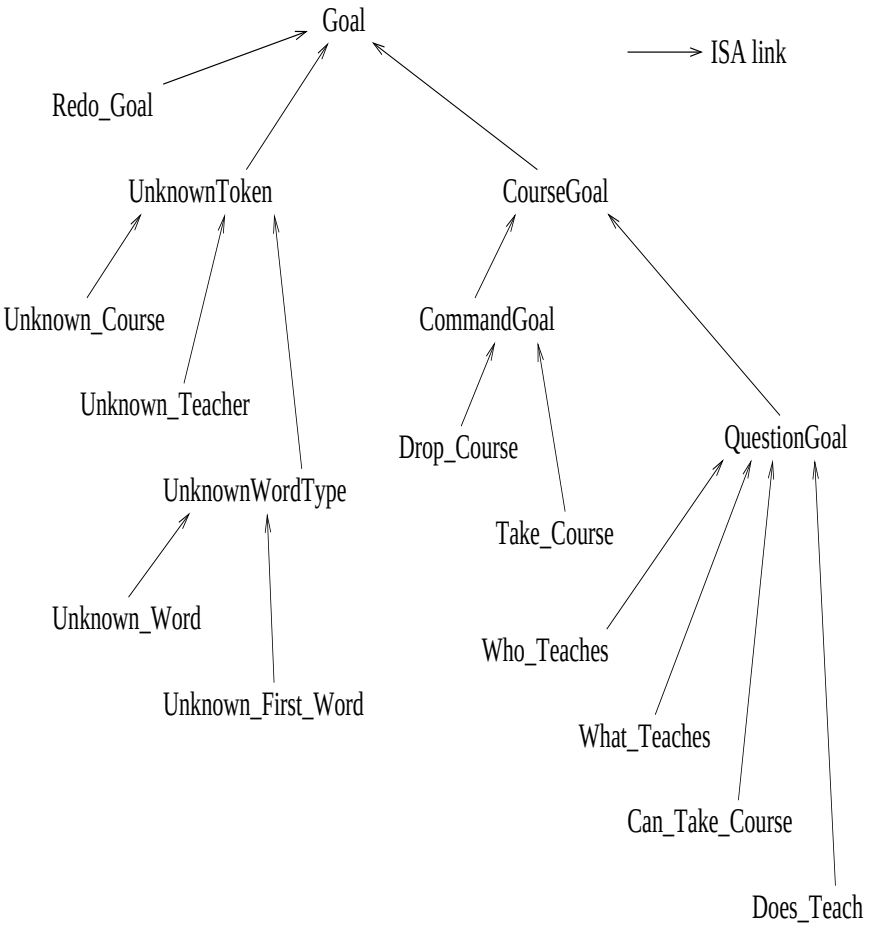


Figure 6.7: Class hierarchy for SG. All the leaf classes (nodes) are concrete, and all the interior classes are abstract. The interior node UnknownWordType was added during the course of implementation, when it became evident that its subclasses Unknown_Word and Unknown_First_Word shared much common structure. The only reason for this node is to store the common functionality that its two subclasses share, and logically UnknownWordType is not needed in SG.

Goal Types and the Penalty Matrix

The penalty matrix shown in Figure 6.10 was developed by considering the consequences of the possible goal orderings; the values in the table will now be explained in detail. Given an ordered pair of goals of type $< A, B >$, A corresponds to the row and B the column of the associated entry in the penalty matrix, and the penalty value is $p(A, B)$. If $p(A, B)$ is less than $p(B, A)$, then the ordering $< A, B >$ is preferred over $< B, A >$; if $p(A, B) = p(B, A)$, then neither ordering is preferred. If A and B are of the same type, then $p(A, B) = 0$ and $p(B, A) = 0$. (To distinguish between goals of the same type in this framework would require one to create new sub-goal types and extend the penalty matrix accordingly.)

We now explain the goals and penalty matrix of Figure 6.10 row-by-row.

Redo_Goal In this row, all the entries are 0, and all the column entries (not on the diagonal) are 1, meaning that if there is ever a Redo_Goal in memory, it should be preferred over every other goal and executed first. A Redo_Goal is meant to be posted when the system gets so confused that its only recourse is to start over from scratch. As shown in Figure 6.11, a Redo_Goal also dominates every other goal, so once it is executed working memory will be wiped clean and the agent will start over anew. Redo_Goals turned out to be useful in the development of *SG* since, if it was ever unclear what an agent should do, a Redo_Goal could be posted which would cause the agent to admit to the user that it was completely confused and will be starting the interaction over again. In its current state, *SG* does not ever post any Redo_Goals, since it can handle all situations by presenting a user with a menu of all the possible ways that they can continue. However, Redo_Goals could easily again become relevant in an extended version of *SG*, so they have been left in. One problem with the Redo_Goals is that they are a sledge-hammer, and make

no effort to find the real culprit in a confusion. One way to extend the notion of a Redo_Goal would be for it to more carefully analyze the state of its confusion to see if it can find the cause, and then instead of deleting all goals in working memory, it will delete/alter only those relevant to the confusion.

The rest of the discussion of the penalty matrix that follows will not be concerned with Redo_Goals, since they are preferred in any set of goals (and are irrelevant in the current version of *SG*).

Can_Take_Course A Can_Take_Course goal is posted whenever the student asks a question like *Can I take cs100?*. *SG* looks at the first token of the user's input, and if it is *can*, *SG* assumes the user is asking this type of question, no matter what tokens follows it. Once it recognizes *can* as the first word, it then looks for the name of a course that follows; if it cannot find a token of the form "csXXX", it then presents the user with a list of all possible courses and asks them to select the one they are asking about. If there is a valid course-token, but it does not correspond to the name of any offered course, then an Unknown_Course goal is posted, which is preferred over Can_Take_Course because it is a sub-goal of that goal. When a Can_Take_Course goal is executed, it checks if the user is allowed to take the course: it makes sure the user has not taken it before, and that they have satisfied all the requisite constraints (e.g. taken all prerequisites, none of the antirequisites, and all of the corequisites).

As shown in Figure 6.10, goals of type Unknown_Course and Unknown_First_Word are preferred to relevant Can_Take_Course goals. This is because before the system can determine if the user can take a course, it must know the first word of an utterance and what course they are asking about.

Does_Teach A Does_Teach goal is triggered by the word *does* appearing as the first

input token, and it deals with questions like *Does Smith teach CS100?*. When executed, this goal queries the course database to determine its answer, and then prints out an appropriate response depending on the results.

As shown in Figure 6.10, a Does_Teach goal should not be preferred to relevant Can_Take_Course, Unknown_Course, and Unknown_First_Word goals. As currently implemented, a Does_Teach and a Can_Take_Course will never appear in *SG*'s working memory at the same time because they are triggered by the (unique) first token in a user's input. However, in an expanded version of *SG* that can handle more than one kind of query/command per turn, then this ordering of these two goals would be relevant.

Unknown_Word *SG* has a lexicon of known words, and for any word that does not appear in that lexicon, it will post an Unknown_Word goal. Unknown_Word goals have the lowest priority and are dominated by all the other goals (Figure 6.11), and so in the current implementation of *SG* Unknown_Word goals are never executed. This allows *SG* to handle input such as *can I tak cs330?*, which is well-formed except for the misspelling of *take*; since all that *SG* cares about is the first word *can*, and that the utterance contains a well-formed course *cs330*, it is not necessary to ask the user what word they meant by *tak*, since it will make no difference to the final result. In contrast, if *SG* does not recognize the first input token, it will then post an Unknown_First_Word as described below.

As shown in Figure 6.10, any goal is preferred over an Unknown_Word goal because, as currently implemented, *SG* only needs to know the first token of an utterance, and if that utterance contains a course or teacher name. All other words, known or unknown, are irrelevant. In a more sophisticated version of *SG*, e.g. one that uses a full-blown natural language parser, recognizing general Unknown_Word goals will

likely be of greater importance.

Unknown_Course If an input token is of the form "csXXX", where each of the X's is a digit, then it is in the correct course name format. If the course is not the name of an actual course, then an Unknown_Course goal will be posted, and executing an Unknown_Course goal will cause *SG* to ask the user to select a course from a menu of all courses it knows about.

As shown in Figure 6.10, only an Unknown_First_Word goal is preferred to an Unknown_Course goal. This is because if the system does not know the first word of the input, it cannot determine what type question/command the user has issued.

Unknown_First_Word An Unknown_First_Word goal is posted when the first input token is not on the list of possible first-token goals. When executed, it presents the user with a list of all possible questions/commands, and asks them to pick one.

As shown in Figure 6.10, an Unknown_First_Word goal is preferred over any goal in working memory (except, as described above a Redo_Goal). This is the case because *SG* determines the type of an utterance by examining the first word in the input.

Take_Course A Take_Course goal is triggered by the word *take* appearing as the first input token, and it deals with commands like *take cs100*, which adds the course CS100 to the user's schedule. A user cannot take a course they have taken in a previous term. Figure 6.8 gives some examples.

As shown in Figure 6.10, a Take_Course goal is only preferred over Unknown_Word and What_Teaches goals. In *SG*, a Take_Course and What_Teaches goal will never appear together (since they both depend on the first word of the input utterance).

Drop_Course A Drop_Course goal is triggered by the word *drop* appearing as the first

```
User> take cs100
You have previously taken CS100, so you cannot take it again.
Courses selected so far: []

User> can i take cs486
Yes, you can take CS486, since it is offered this term.

User> take cs486
Okay, adding CS486 to your schedule.
Courses selected so far: [course(CS486)]

User> take cs486
You have already put CS486 on your schedule.
Courses selected so far: [course(CS486)]
```

Figure 6.8: Examples of the take and drop command in SG.

input token, and it deals with commands like *drop cs100*, which removes the course CS100 from the user's schedule. When executed, checks the student's schedule to see if CS100 is on it, and then prints out an appropriate response depending on the results.

As shown in Figure 6.10, a Drop_Course goal is only preferred to Unknown_Word, Who_Teaches, and What_Teaches goals, although, like some other goals above, a Drop_Course goal and Who_Teaches, and What_Teaches goals cannot appear together at the same time.

Who_Teaches A Who_Teaches goal is triggered by the word *who* appearing as the first input token, and it deals with questions like *Who teaches cs100?*. Figure 6.9 shows an example. When executed, this goal queries the course database to determine its answer, and then prints out an appropriate response depending on the results.

As shown in Figure 6.10, Does_Teach, Unknown_Course and Unknown_First_Word

```
User> who teaches cs1000

You asked a 'who' question, but gave no course.
Please select one of the following courses:
  0-CS494      1-CS488      2-CS486      3-CS476
  4-CS466      5-CS462      6-CS454      7-CS452
  8-CS448      9-CS446     10-CS445     11-CS432
 12-CS430     13-CS370     14-CS370     15-CS360
 16-CS354     17-CS351     18-CS342     19-CS340
 20-CS338     21-CS330     22-CS246     23-CS246
 24-CS240     25-CS240     26-CS240     27-CS230
 28-CS212     29-CS200     30-CS134     31-CS134
 32-CS134     33-CS130     34-CS130     35-CS130
 36-CS130     37-CS130     38-CS120     39-CS120
 40-CS102     41-CS100
Enter number of selection (0-41): 1

Okay, 'CS488' selected
CS488 is taught by [MANN]
```

Figure 6.9: Example of a *who* question. The user has typed a valid question, but the course "cs1000" is invalid, so the system displays a list of all the possible courses (duplicates are caused by multiple teachers of the same course) and asks the user to select which one they meant (the user types in the number 1, indicating they are asking about CS488). After this selection is made the system can then satisfy the original Who_Teaches goal.

goals are preferred to a Who_Teaches goal; and Who_Teaches goal and Does_Teach goal cannot occur together at the same time.

What_Teaches A What_Teaches goal is triggered by the word *what* appearing as the first input token, and it deals with questions like *What courses does Smith teach?*. When executed, this goal queries the course database to determine its answer, and then prints out an appropriate response depending on the results.

As shown in Figure 6.10, only an Unknown_First_Word goal is preferred to a

	Redo_Goal	Can_Take_Course	Does_Teach	Unknown_Word	Unknown_Course	Unknown_First_Word	Take_Course	Drop_Course	Who_Teaches	What_Teaches
Redo_Goal	0	0	0	0	0	0	0	0	0	0
Can_Take_Course	1	0	0	0	1	1	0	0	0	0
Does_Teach	1	1	0	0	1	1	0	0	0	0
Unknown_Word	1	1	1	0	1	1	1	1	1	1
Unknown_Course	1	0	0	0	0	1	0	0	0	0
Unknown_First_Word	1	0	0	0	0	0	0	0	0	0
Take_Course	1	1	1	0	1	1	0	1	1	0
Drop_Course	1	1	1	0	1	1	1	0	0	0
Who_Teaches	1	0	1	0	1	1	0	0	0	0
What_Teaches	1	0	0	0	0	1	0	0	0	0

Figure 6.10: Concrete goal penalty matrix. Only 1's and 0's appear in this particular matrix.

What_Teaches goal.

Managing Working Memory in *SG*

Working memory is implemented as a vector of goals, and can contain any number of concrete goals.[3] Goals are then ordered using one of the local search algorithms from Section 4.5. Even for a large number of goals, the real time performance of these algorithms is good. Every time a goal is modified or added to working memory, the ordering algorithm is run again. Deleting a goal from working memory is trickier than adding one. First, goal A *dominates* goal B if satisfying A makes the satisfaction of B superfluous. For example, if the system has the goal to ask the user to enter her name, but the user happens to tell the system her name without being prompted, then the system should

[3]New concrete goals can be added to *SG* and, thanks to the object orientedness of C++, will work with the working memory code without modification. Limits on the number or type of goals that working memory can contain can easily be added, if desired, since all access to working memory is through a single class interface.

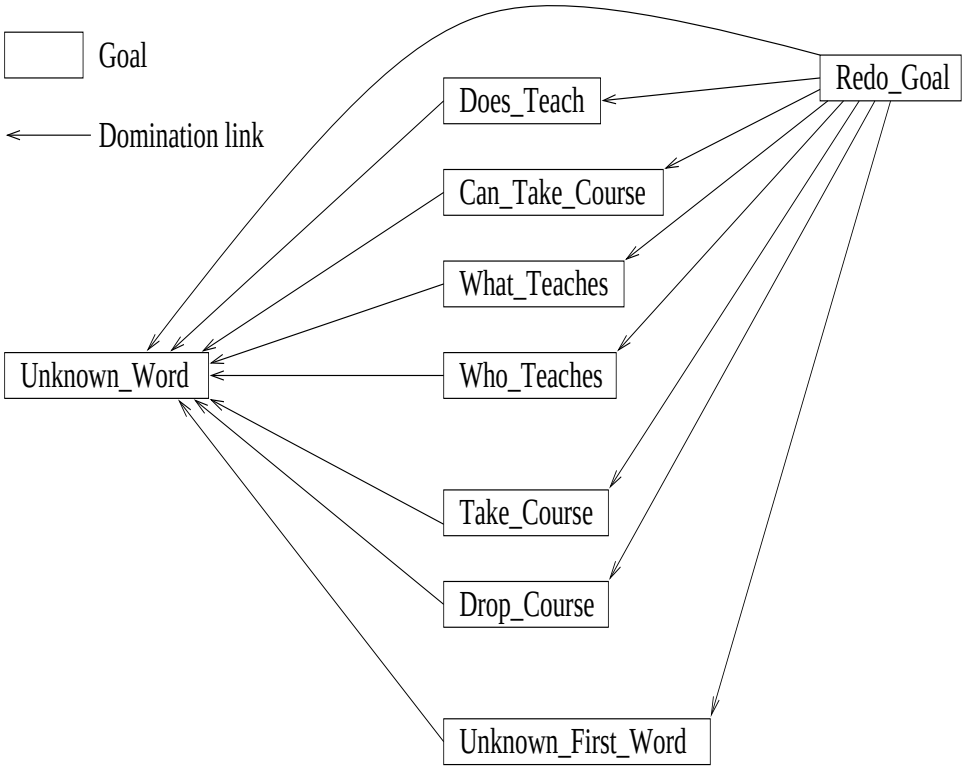


Figure 6.11: Goal domination hierarchy.

not try to satisfy the name-getting goal. In SG, if A dominates B and A is satisfied, then A is removed along with B (and any other goal A dominates). Domination is in terms of goal type, and goal time/content. Figure 6.11 shows what goal types dominate each other.

Repairing Utterances

An interesting feature of SG is that it automatically repairs expressions containing mistakes. Suppose the user asks SG this question:

Dialog 6.1

User: Can I take cs130

From a logical point of view, this utterance is equivalent to evaluating $can\_take\_course(cs130)$. SG has special code for handling each kind of predicate, so this can be thought of as mapping to a procedure call with course parameter $cs130$. In this case, all is well because $cs130$ is a known course, and makes sense in this predicate. However, suppose the user asked this question:

Dialog 6.2

User: Can I take cs133

Because $cs133$ is not a known course, two goals corresponding to the following predicates are posted: $can\_take\_course(cs133)$ and $unknown\_course(cs133)$. Because repair goals[4] have priority over non-repair goals, $unknown\_course(cs133)$ will be executed first and will not terminate until it gets a known course from the user; it does this by showing the user a list of all known courses, and asking him to select one.[5] Supposing the newly selected course is $cs130$, this must replace $cs133$ in $can\_take\_course(cs133)$ (and any other relevant goals). Clearly, it would not be hard to scan working memory and make the substitution in each goal, but SG takes advantage of an implementation detail to make this change in constant time. The goal data structure keeps a *pointer* to its parameter(s), and if two or more goals have the same parameter, they point to the same single parameter object. Thus, when changing $cs133$ to $cs130$, only one parameter object is changed, affecting all the goals that point to it; see Figure 6.12.

[4]The repair goals are Unknown_Word, Unknown_First_Word, and Unknown_Course. As previously explained, it turns out that it is never necessary to execute an Unknown_Word goal, because the other goals take care of all the problems that arise. Thus, in this exceptional case and in contrast to the other two repair goals, Unknown_Word has a low priority.

[5]The implementation currently does not allow for the user to "bail out" of such a situation. It is straightforward to add this feature.

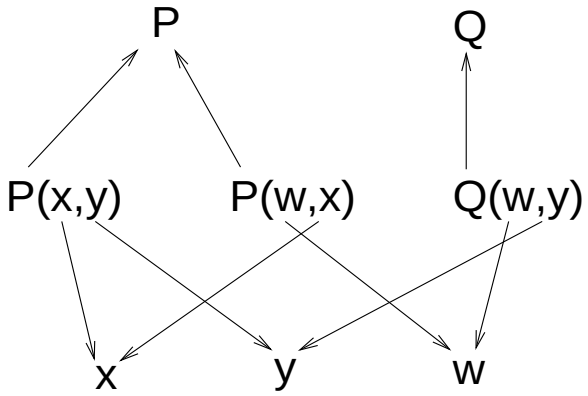


Figure 6.12: The structure of goals. It is unrealistic to assume that the user will always enter error-free input. In *SG*, goal objects contain pointers to their parameters (but not their predicates, unlike this example), so if, say, w is changed to v, then automatically $P(w, x)$ and $Q(w, y)$ become $P(v, x)$ and $Q(v, y)$. The same trick can be used with predicate names, although it leads to complications such as handling predicates with different arities.

6.4.1 Spelling Correction

> . . . I don't think it's quite fair to condemn the whole program because of a single slip-up. . .
>
> *General Buck Turgidson, Dr. Strangelove*

SG is able to spell-check user input, which turns out to be a fairly useful feature. *SG* uses a simple "reverse edit distance" spelling correction algorithm (Kukich 1992): if *SG* comes across a word not in its dictionary, it applies a series of letter-transformation algorithms to the unknown word, and any transformation that results in a known word is suggested to the user as a possible correction. The transformations on the unknown word include: deleting single letters, adding single letters, and swapping adjacent letters. This method provides good suggestions for many errors that contain essentially one mistake. For example, if the unknown word is "afternoonk", then at least "afternoon" would be suggested, i.e. removing "k" from "afternoonk" results in a known word. The methods used by *SG* would not suggest the correct spelling if the user made two mistakes and

type "wafternoonk"; however, a single-error corrector would suggest the correct words for "wafter noonk", since these are treated as separate words with one error each. Also, the spelling corrector cannot catch words misspelled as other words, e.g. "user" can accidently be typed as "suer". Catching this type of error requires some knowledge of the syntax and semantics of language.

6.4.2 *SG* and Turn-taking

As implemented, *SG* does not explicitly embody the entire turn-taking model, although it does solve the next-action problem using local search algorithms (Section 4.6). Since it is a command-line system, *SG* interprets the user pressing the return key as an explicit turn-ending signal. In this section, we will discuss how *SG* could be modified to more fully account for turn-taking by analyzing it in terms of the three-step turn-taking model (Chapter 3).

Before doing this, though, we note that a command-line system such as *SG* is not the ideal way to demonstrate the value of turn-taking. A major asymmetry between users and such type-written systems is that, while they both may take a comparable amount of time to think about what to say and how to word an utterance, computers can "type" out their utterances much faster than humans. In *SG*, the human must type all their input at the keyboard, while the system (essentially) displays its output by calling print statements, which take significantly less time to execute. This means that there is no natural way for the user to interrupt the system in mid-turn because the output is displayed in a fraction of a second, although it is quite possible for a system to interrupt the user in mid-turn. Because of this asymmetry, we will not consider the possibility of the user interrupting the system.[6] The greatest value and need for a system that knows

[6] *SG* could be re-implemented along the lines of the Unix `talk` command, where the system and user get their own windows to type in, and either can type at any time. This is perhaps a better style of

how to take turns will likely appear in spoken dialog systems, and prototypes such as *SG*, while admittedly imperfect, do avoid the numerous implementation complexities of spoken language systems, which are significantly different than linguistic theories and systems geared for written language (Section 2.5).

Motivations in *SG*

Goals in *SG* are triggered by a number of well-defined circumstances: the first word of an utterance, unknown words, and unknown course names. *SG* treats the user's input as a list of tokens, and scans it looking for goal-triggering conditions; any recognizable circumstances can be used to trigger goals, i.e. a goal recognition can execute arbitrary C++ code in its operation. Of course, one should take into consideration the fact that conversation is ultimately a real-time activity, or at least a time-pressured one.[7] As mentioned, *SG* does not use full-blown natural language parsing in part because parsing is a significantly complicated activity that can cause time delays, so it would be necessary to use a parser that is either unusually efficient, or is somehow sensitive to real-time performance. To avoid the inherent implementation difficulties, and given that this thesis does not go into the motivation step beyond what is covered in Section 3.4, *SG* settles with using a simple but efficient keyword-based technique (although see Section 6.4.3, where it is argued that this is not a completely implausible approach to language understanding). Extending this step to do resource-sensitive syntactic or semantic interpretation would be an interesting and likely non-trivial project (Section 8.2).

---

implementation, but it is still imperfect, since, as experience with the talk command shows, it is possible for users to concentrate on their own typing and not realize the person they are speaking to is also typing. Since typing in different windows at the same time is not as serious a problem as speaking (through the same channel) at the same time, the situation is not completely analogous to spoken dialog.

[7] The user might tolerate some stops and starts, but numerous long or arbitrary-seeming pauses are likely to cause frustration. World Wide Web browsers, for example, sometimes suffer from such behaviour, due to network and processing delays which cause the browser to start and stop in a way that many users find quite annoying.

Processing in *SG*

As already discussed, *SG* decides what to say next by using local search algorithms to solve the next-action problem; the implementation closely follows the specifications given in Chapter 4.

Turn-taking Logic in *SG*

SG does not directly implement the turn-taking logic or protocols of Chapter 5, so here we will discuss how *SG* could be extended to take turn-taking explicitly into account.

utterances

user User utterances consist of what the user types at the command-line prompt. Sometimes *SG* asks the user to select from a menu, and the user enters a number; this is a kind of utterance, but it is much more constrained than the sort of utterances the user enters at the top-level prompt because only a small range of numbers are accepted as legal input. *SG* will continue displaying the menu until the user makes a legal choice, so this forms a menu-selection subdialog where the turn-taking is completely regimented, and could reasonably remain so in more sophisticated implementations, i.e. the system allows full-blown natural language input at the top-level, but then initiates more constrained subdialogs in particular circumstances (for example, the CvBS system described in Section 7.5 does this).

SG System utterances consist of displaying text on the screen, including replies to user queries/commands, and listing menus of options for the user to select from.

back-channel utterances

user The user cannot make any kind of back-channel utterance in the current implementation of *SG* (due partly to the asymmetry mentioned above). In a more sophisticated version of *SG*, one could imagine the user issuing the same sorts of confirmations that appear in human-human conversation. For example, if one person is speaking for an extended period of time, the listener can indicate they are following and paying attention by issuing brief back-channel utterances like *okay*, *uh-huh*, *alright*, etc. Also, in cases of not understanding something the speaker has said, then sometimes the listener can indicate this problem through a back-channel utterance such as *sorry?*, *huh?*, *uhhh*, etc. It would be somewhat comical to require a user to type such phrases, but in a speech-based dialog system such feedback can be useful, and would occur naturally. In *SG*, or similar command-line interfaces, another technique would have to be used. Suppose a more sophisticated version of *SG* generates extended multi-line utterances in such a way that there is a reasonably long but natural pause between sentences/clauses. During such pauses, the system could present a "confirmation" menu to the user, allowing them to enter "yes", corresponding to a confirming back-channel utterance like *okay*, etc., or "no" corresponding to a back-channel like *uhhh*. If such delays do not exist to make such a technique feasible, then another possibility is for *SG* to label each part of its output, for example giving each sentence a unique identifying number. The system prints out its response with the sentences so labelled, and as the user reads, they can give feedback to the system as they read by referring to the sentence numbers. Of course, this will not affect how any of the already printed output is displayed, but it does allow for the system to elaborate or re-state parts the user has trouble with, plus it gives the user a chance to avoid reading all of a long utterance if, say, halfway through they realize the

computer's response is not relevant. The user could issue a command indicating they have stopped reading after a certain sentence number, so the system could behave as if it never printed out anything past that sentence, which in a way simulates an interruption in spoken dialog.

SG *SG* currently does not issue any back-channel utterances, but it could be augmented to "listen" as the user types, and give confirmations like those discussed above in the user case. Of course, we do not want the system to output text using the same cursor the user is typing with, so a second channel of communication would work best, e.g. a second window, auditory feedback (e.g. beeps or tones), or visual feedback, such as turning the text the user is typing to a different color (e.g. green means "okay", and red means "confused"). *SG* currently does allow for some degree of incremental interpretation, and it would be a straightforward matter of programming to add back-channel feedback to *SG*; the bigger challenge would be to do it in a useful, non-disruptive way, which would likely require trials with numerous users.

turn-taking signals

user

turn-ending signal Currently, pressing the return key is a turn-ending signal from the user. Pressing return causes *SG* to begin to process and interpret the user's utterance, although it is just an implementation limitation that *SG* does not do any processing while the user is typing. Using the return key as a turn-ending signal is quite reasonable and common, and in the culture of command-line interfaces it is commonly understood that pressing the enter or return key is how things get done.

turn-yielding signal *SG* does not recognize any turn-yielding signals in the user's input, although it is interesting to consider how they could be added. A turn-yielding signal is like an abstract punctuation symbol that warns of an upcoming turn-ending signal. Since the user is typing in English, some of the features that indicate a turn-yielding signal in English conversation could be watched for (Oreström 1983), or one could imagine applying a learning algorithm to look for features of the user's input that usually occur before they press return. Such features might include punctuation, grammar, or, possibly, even typing speed — there is no necessary reason to limit turn-yielding signals to the kind that occur in human-human conversation. The value in recognizing a turn-yielding signal is that the system has some advanced warning about when it will be required to reply, and this warning provides it some information that can help it manage its processing (e.g. it should probably not begin a lengthy calculation).

interruptions Interruptions share some of the same concerns discussed with respect to user back-channel utterances. A user interrupts the system when the user takes the floor in a manner that does not follow the turn-taking protocol. Currently, interruptions are not possible in *SG*, and as previously discussed, the asymmetry in human/computer typing makes it very difficult for the user to interrupt *SG*. However, one natural place to allow the user to interrupt is when *SG* presents the user with a menu of choices, and does not continue until the user has selected one of the choices. In some cases, it would be preferable to jump back to the top-level of interaction in *SG*, which would be an interruption in the sense that the menu subdialog is an exceptional and relatively uncommon action. Interruptions could also be used in one of the alternate schemes

suggested in the section on user back-channel utterances. For instance, if each output clause is labelled with a number, the user could read the output and interrupt by "inserting" their utterance before a particular numbered clause. The system would then need to behave as if each clause after the interruption did not occur, since the interruption could change the course of the dialog.

Since computers do not get frustrated the way humans do, there is less concern for indicating the best times for users to interrupt the system. In theory, it is possible for a program to capture and save the current state of its computation so it can be re-started at a later time. It might be unreasonable for a program to be interrupted in particular sub-parts of its program, but beyond such constraints, a user should be able to interrupt at any time.[8]

SG

turn-ending signal *SG* signals the end of its turn by displaying the top-level command prompt, or by asking the user to make a selection from a displayed menu. This is reasonable turn-ending signal, although one can imagine augmenting it with visual or auditory cues to help let the user know when it is their turn to speak.

turn-yielding signal A turn-yielding signal from *SG* is meant to be interpreted by the user as a sign that the system is about to end its turn. Turn-yielding signals play a similar role as they do in the user case above, giving the user extra information that may influence their thinking, and

[8] Chalmers (1996) presents interesting arguments suggesting that any system with the right functional organization will have conscious experiences. If this is correct, then a computer may indeed experience frustration from untimely and pointless interruptions. But, since such frustrations would not change the behaviour of the program, they would be irrelevant to computer science.

also avoiding sudden turn-ending signals from the system which the user might not notice, or could find jarring. As with interruptions, the speed at which *SG* displays its output limits the usefulness of explicit turn-yielding signals.

interruptions *SG* has more natural opportunities for interrupting the user than the user has for interrupting *SG*. In the course-advising domain, there does not appear to be any sort of utterance so important that it should be made without regard to the user, i.e. interrupting the user in mid-speech is not usually a good idea. In riskier domains, some utterances might be so important that their importance alone warrants an interruption at any time; for example a intelligent navigation assistant on an airplane might interrupt a pilot to warn of an unforeseen obstacle, a sudden change in the weather, etc., since knowing about that is more important than almost anything else the pilot might be doing. But in course-advising, any utterance can wait until the end of a turn, and there is nothing of such knock-down importance to worry about, so interruptions are never vital. But interrupting can be useful, for example in cases where, if the system does not interrupt, then the user might waste a lot of time typing in irrelevant facts. For example, the student might begin to tell the systems their educational "life story", when all the system needs to know is what subject they are majoring in. The earlier that such an interruption is made, the more wasted effort the system will prevent, so such an interruption should be made as soon as the system realizes what the student is doing. The system might believe that the student's academic life story will contain the information it needs, and so performing such an interruption could be based on a general interaction principle, e.g. a collaborative agent should

not let another agent do what it knows to be irrelevant work. However, user testing would seem necessary here, since it is not completely obvious that the time saved by interrupting a user in this way is worth the annoyance it may cause the user, who might actually enjoy telling the system their educational life story, and so perceive the interruption as rude or unfriendly.

disruptions Disruptions (interruptions by nature) do not occur in *SG*, although they can certainly occur in real-world advising (Section 5.3.6). *SG* could be extended, for example, to handle interruptions from the operating system, related to email or other processes running on the computer.

6.4.3 The Plausibility of the Keyword Approach

Grasshopper always wrong in argument with chicken. *Book of Chan*

In the plaza next to the University of Waterloo, a fast-food restaurant sells two kinds of chicken sandwiches: the supreme *grilled chicken*, and the *crispy chicken*. If one orders just a "chicken sandwich", then the employee must ask if you mean a crispy chicken or a grilled chicken sandwich. Presumably, this is common enough that there is a "take an order for a chicken sandwich" script that employees follow, which essentially requires them to ask *Do you want a grilled or crispy chicken sandwich?*.

This first example is based on a true encounter at a restaurant that makes a similar distinction:[9]

[9]From Dilbert's Newsletter 17.0, August 1997.

Dialog 6.3

Employee: Crispy or regular?

Customer: I don't care. Either will be fine.

Employee: [annoyed] Crispy or regular?

Customer: ...Ahh, Crispy then.

Employee: We are out of crispy.

The problem here is that the employee is requiring the customer to make a choice when in fact there is no choice; only grilled chicken sandwiches are available. This is clearly non-optimal behaviour on the part of the employee, although perhaps the employee is just thoughtlessly following the regular "take an order for a chicken sandwich" script.

The following example, which actually occurred to the author of this thesis, is also instructive:

Dialog 6.4

Customer: I'd like a crispy chicken sandwich please.

Employee: Do you wanted a grilled chicken sandwich, or a crispy chicken sandwich?

Customer: I'd like a crispy chicken sandwich please.

Employee: Do you mean the deep-fried chicken sandwich, or the grilled one?

Customer: [pointing to menu] I'd like a crispy chicken sandwich please.

Employee: [confused] Okay.

The customer made a point of explicitly naming the kind of chicken sandwich he wanted, since he was familiar with the hazards of ordering food at such restaurants as illustrated by the previous example. In this case, it appears that during the customer's first request the employee realized he was ordering a chicken sandwich, and without listening to the entire utterance, but, perhaps out of general politeness, waiting until he had finished speaking, began following a "take an order for a chicken sandwich" script. After repeating the same order, the employee seemed to realize that the customer had in fact "properly" ordered the first time, and that she had not listened to his exact order. Thus, she apparently came to believe he ordered properly both times, but was unsure if he had made the same order each time. So her third utterance could be seen as seeking to clarify this potential ambiguity. The whole problem seems to have been caused by the employee committing too quickly to the "take an order for a chicken sandwich" script, e.g. her course of action was determined as soon as she heard the word "chicken".

ELIZA

> At least in Byblow Bottom when we talked, it was on subjects relevant to our life ... But here in Bath, conversation was, it seemed, an end in itself. Materials for it were collected like kindling wood, but then used in an artificial and prodigal manner, merely to generate a flame. But to what end? Simply to make a sound, to fill a silence, to pass a period of time. Time which, it seemed to me, could in a thousand ways have been more profitably spent. *Joan Aiken, Eliza's Daughter*

ELIZA is one of the most famous programs in artificial intelligence. It was developed by Weizenbaum (1966), and has since become a mainstay "play toy" available on most computer systems, and is often set as a programming exercise for computer science students. ELIZA was never intended to be a complete natural language processing system, but instead relies on simple keyword matching to produce dialog that is, sometimes, remarkably realistic. Figure 6.14 and Figure 6.15 show sample ELIZA dialogs.

ELIZA works essentially by keyword matching. For example, the implementation described in Norvig (1992) is a surprisingly brief one, consisting of little more than a rule-based pattern matching system. ELIZA takes input from the user and scans through a list of the following kinds of rules:

```
?x I want ?y --> What would it mean if you got ?y
                 Why do you want ?y
                 Suppose you got ?y soon
```

The left-hand side of the rule specifies a text pattern to match in the user input. If the match is made, then one of the responses on the right-hand side of the rule is chosen and displayed as output to the user (with the appropriate variable substitutions made). And that's about it. Norvig's implementation always picks a random response from the right-hand side of the first rule that matches, but many variations are possible. When no rules match, ELIZA can pick from a list of canned phrases designed to elicit more information, or, in some versions, ELIZA remembers user input in order to refer back to it later. Figure 6.13 gives some examples of the types of heuristics ELIZA-style programs incorporate.

All of ELIZA's conversational ability is implicitly encoded in its rule set. ELIZA does not understand the meaning of words or sentences, so it cannot make even gross distinctions between sensible input and gibberish, and will often repeat spelling errors and other mistakes the user makes. It does not use any basic natural language understanding or generation techniques found in most other natural language systems (Allen 1987).

While ELIZA is not a good model of a general-purpose natural language agent, there is something to be said for the efficacy of its behaviour given the simplicity of its design. ELIZA has a minimal amount of knowledge, and yet sometimes appears to be having meaningful conversation. Because it does not understand anything, ELIZA is never fooled

1. agree with the user (as opposed to be non-committal)
2. make utterances that encourage users to talk about themselves
3. admit ignorance if the answer to a question is not known
4. make utterances that encourage the user to change the subject of the conversation
5. make controversial statements
6. make humorous statements

Figure 6.13: Some of the heuristics used by Mauldin (1994) for the 1993 Loebner Contest. Other heuristics are used, but the point of some are unclear, as are their actual efficacy and relative utility.

USER: I have trouble setting career goals.

ELIZA: What makes you believe you have trouble setting career goals?

USER: I can never keep a job.

ELIZA: Is it because of the people you hang around with that you say you can not ever keep a job?

USER: Yes, definitely. I find it hard to make friends who don't ignore me.

ELIZA: Is it because of problems at school that you are going through all this?

USER: My math classes are giving me nightmares.

ELIZA: Tell me something about math.

USER: I hate it, and wish I'd never gotten involved with it. I wish I had taken drafting instead.

ELIZA: Are you sure about that?

Figure 6.14: Sample ELIZA transcript. This was generated by a user interacting with the "doctor" program available on the EMACS text editor; the user intentionally tried to steer the conversation towards coherent output, and the results give some idea about how life-like ELIZA can seem. However, compare this output to that of Figure 6.15.

USER: What time is it?

ELIZA: What makes you believe what time is it?

USER: I need to buy a new computer.

ELIZA: What makes you believe you might want to buy this new?

USER: Is love all you need?

ELIZA: Perhaps you could tell me about your relationship with love all i.

Figure 6.15: Another sample ELIZA transcript. This was generated by a user interacting with the "doctor" program available on the EMACS text editor; the user intentionally tried to steer the conversation towards incoherent output, and the results show the shallow nature of ELIZA's understanding.

or confused by inconsistent input the way more intelligent programs are (particularly ones that encode knowledge about some particular domain of discourse). While people surely do not always process the language the same way ELIZA does, ELIZA may actually provide a reasonable explanation of how people process language *some* of the time. For example, in the chicken-sandwich dialogs in Section 6.4.3, the cashier's behaviour can be explained with scripts and keyword matching. In other cases, individual words can influence how some people interpret a conversation more-so than the content, e.g. if a speaker uses racial slurs, the listener might disregard what they say. Or, one might, for example, avoid mentioning "baseball", since that word causes some people to begin talking exclusively about baseball.

The *SG* system, while much more sophisticated than ELIZA (at least in the course advising domain!), uses a kind of keyword matching on the user input in order to avoid the complexities of parsing natural language.[10]

[10]The Loebner Contest has attempted to advance the cause of AI by having human judges distinguish between human and computer conversationalists in a manner similar to the Turing Test. Part of the

6.5 SAM: The Simple Advising Model

SAM is a Java-based graphical system that presents a very different way of dealing with the course advising model. It allows the student to construct a table of courses on the screen by selecting courses and placing them in a graphical representation of the schedule. Through the use of a constraint-checker,[11] the system will not allow the user to create selections that contain any hard constraints. SAM also allows for a simple user model in terms of a list of courses that the student has taken, and so cannot take again. This design makes a clean distinction between system knowledge and user knowledge: the system has the database of courses, and knows when all the courses are offered, how many courses a student can take during the day, etc. The choices the student can make correspond to preferences that will likely vary from user to user.[12] The hard constraints are designed into the interface, and it is impossible for the user to do anything to violate them. For example, a course can only be selected once, so once a user picks a course and adds it to the schedule, it is no longer selectable;[13] SAM also automatically makes unselectable any courses that cannot be taken at the same time as an already selected course. Figure 6.16 shows a screen-shot of SAM.

A disadvantage of this GUI approach is that there is often no obvious way to allow the user to ask informational questions, or to find out why particular schedules are disallowed.

---

contests notoriety comes from the fact that most of the entrants are modified versions of ELIZA (e.g. Mauldin 1994), which makes the contest of little interest to most AI researchers, who have long since abandoned such techniques. Shieber (1994) provides a critique of the first Loebner Contest from the point of view of a mainstream AI researcher.

[11] A constraint-checker simply determines whether a set of values assigned to CSP variables (see Section 4.3.1) is allowable or not. It cannot solve a CSP, but is implicitly part of any CSP solving algorithm. In SAM, a constraint is either satisfied or unsatisfied, but a constraint-checker could return a degree of satisfaction, as in a COP (Section 4.3.1).

[12] Since this distinction is hard-coded, then according to the distinction made in Figure 5.1, SAM is *not* a mixed-initiative system.

[13] This can be nicely implemented using the common GUI technique of "drag-and-drop", although SAM does not use this technique due to infelicities in the early versions of the Java windowing library.

For instance, in *SG*, it would be straightforward to allow the user to ask questions like *Why can't I take cs130 this term?* and have the system give a reasonable and helpful response. But it is unclear how such questions could be easily integrated into SAM without adding extra complexity to the on-screen interface (e.g. extra buttons, menus, sliders, ghosted selections etc.), or resorting to an *SG*-style prompt-line interface for certain kinds of questions. This is not an insurmountable problem, but the direct-manipulation paradigm lacks the flexibility of a command-line interface.

6.5.1 SAM and Turn-taking

To apply turn-taking to SAM, we must show how it fits into the 3-step model (Chapter 3), and how the relevant concepts of the turn-taking model map to it (Figure 6.2). SAM can be seen as instantiating the model in the following ways (summarized in Figure 6.17).

Motivation in SAM

As with CSPs (Section 4.3.1), the constraints on what courses that can appear together in a schedule can be represented as tuples, representing either an allowable set of courses or a conflicting set of courses. For example, if the lectures for CS351 and CS360 overlap, then these two courses cannot appear in the same schedule. In general, given a set of courses $C = \{C_1, \ldots, C_n\}$, and a set of tuples $T_1, \ldots, T_k$, where T_i is a subset of C, a schedule is defined to be a set of courses S such that none of the T_i's is a subset of S. Less formally, the T_i's are sets of courses that cannot occur in the same schedule (due to some sort of conflict between them), and a schedule is a set of courses that contains none of these conflicted subsets. In SAM, the user is allowed to add courses to S, but the system continually checks to make sure that the user is only able to select courses that will not result in a constraint violation, i.e. SAM maintains the "schedule invariant" that no T_i is a subset of S. Treating the interaction as a dialog, SAM's most important action

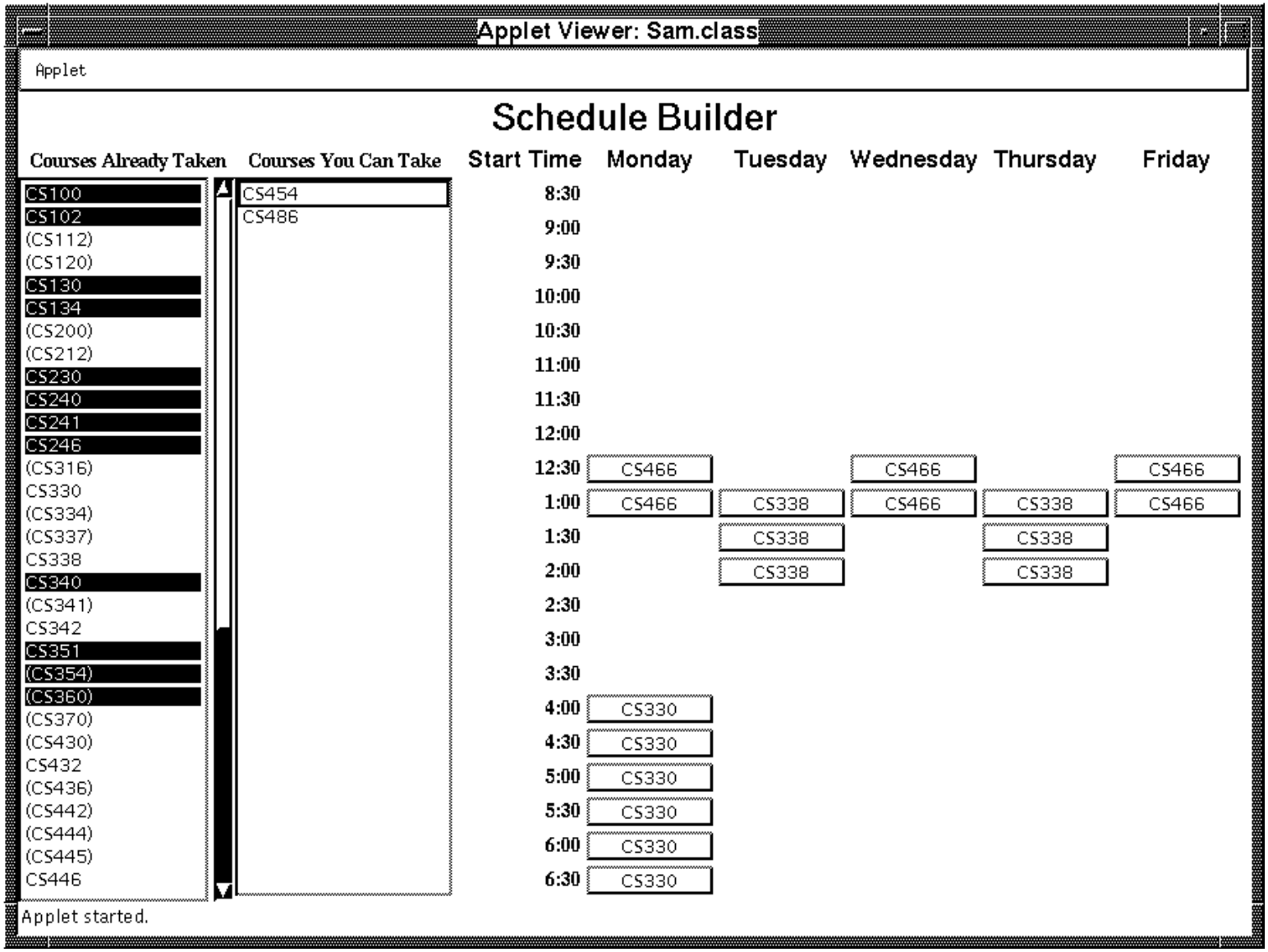


Figure 6.16: Screen shot of SAM. The user makes selections from the right-most box on the left-hand side of the screen, and they are then automatically added to the schedule on the right-hand side. It is impossible to select conflicting courses, as all conflicts are removed as soon as the user selects a course.

Motivation Constraint-like rules, discussed in Section 6.5.1. Simple, efficient, and can allow for incremental processing.

Reasoning Maintenance of the "schedule invariant"; requires only basic set-operations, as discussed in Section 6.5.1.

Turn-taking See Section 6.5.1.

utterances User Mouse-actions. e.g. clicks and movements.

SAM Screen updates, e.g. rewriting the list of allowable courses, changing the schedule, etc.

back-channel utterances User None.

SAM None, but possibilities suggested in Section 6.5.1.

turn-taking signals

User

turn-ending signal A mouse action that causes the set of selected courses to change.

turn-yielding signal Implicitly, moving closer to a screen location where a schedule-change can be made. Other turn-yielding signals might be learnable by analyzing user actions.

interruptions User cannot currently interrupt SAM.

SAM

turn-ending signal Return of control of the mouse-pointer to the user.

turn-yielding signal None.

interruptions None currently, but possibilities for interruptions from SAM are discussed in Section 6.5.1.

disruptions None currently, but possibilities for disruptions are discussed in Section 6.5.1.

Figure 6.17: Turn-taking summary of SAM. This is based on the questions from Figure 6.2.

is to redraw the list of allowable courses such that the schedule invariant is maintained (it must also manage all other aspects of the interface, but this does not influence the schedule invariant). Thus, SAM's turns consist of redrawing the screen, and they are motivated by the user adding or removing a course from S (removing a course from S sometimes allows other courses to be added to S).

Processing in SAM

The basic computational problem SAM must solve is to maintain the schedule invariant. Whenever S changes, SAM must check which courses can now be added to S and hence displayed to the user. Conceptually, SAM must run through all the courses $C = \{C_1, \ldots, C_n\}$ and determine if adding C_j to S will result in a constraint violation, i.e. if $C_j \cup S$ contains any of the T_i's as subsets. It is relatively straightforward to calculate subset membership, and, for example, Cormen et al. (1990) discuss numerous ways to efficiently process sets and subsets. In practice, SAM uses a very straightforward updating algorithm that results in very fast real-time updating,[14] in part because there are not that many courses. What to do if there is a noticeable processing delay due to this step will be discussed below.

Turn-taking Logic in SAM

To describe how SAM instantiates the third step, we must show what the concepts from the turn-taking logic (Section 5.3) map into in SAM. First, a user utterance will assume to be terminated by a mouse-action that causes S to change (through the addition or deletion of a course). Furthermore, such an action will constitute a turn-ending signal. The user may move the cursor around and perform other actions, but they will all be

[14]This is on a Sun Sparcserver 1000. The Java implementations on which SAM was tested were interpreted and so not especially efficient.

lumped into one utterance. SAM only knows about turn-ending signals, in part because its processing is done effectively instantaneously (from the point of view of the user) right after receiving a turn-ending signal, so there is no need for a turn-yielding signal. If the processing could not be done so efficiently, which could happen if there are very many courses and constraints, then a turn-yielding signal should be defined. For example, if the mouse-pointer comes within a certain distance of the region on the screen where the user selects courses, then that could be interpreted as a turn-yielding signal, i.e. an offer of the floor to the system; if the pointer moves out of the region, then that action can be interpreted as the user retracting the offer. Being close to a button (for example) does not mean that a user will press it, so there could be many turn-yielding signals before a turn-ending signal. Alternatively, this could also influence the design of the interface, i.e. force the user to do something before they select a course that can reliably be interpreted as a turn-yielding signal.

SAM's utterances consist of redrawing the screen such that the user cannot make a course selection that violates the schedule invariant. SAM's turn-ending signal returns control of the mouse-pointer to the user, and, as described above, SAM does this so quickly that there is no appreciable processing delay. Thus, again, there is no need for turn-yielding signals from SAM. If the processing took more time, then we would want to be able to monitor the processing algorithm such that the system could issue a message, for example popping up a window showing a progress-bar indicating how close the algorithm is to completion. In a GUI environment, this can be treated as a kind of stalling phrase (Section 5.4.3) that provides somewhat more useful feedback than is usually possible in a command-line system such as *SG*, which would have to issue English phrases.

SAM currently never interrupts the user, but there are a number of ways in which it could. For example, SAM's constraint checker could be extended to a full-blown CSP

solver, and so while the user is thinking about what course to add, SAM could be figuring out allowable schedules for the user on its own. In some cases, it is conceivable that there might be only one legal schedule, hence SAM may as well take the initiative and interrupt the user, displaying the one possible schedule. It would also be possible to use "warning sets", i.e. sets of courses $W_1, \ldots, W_m$ such that if S ever contains a W_i as a subset, then the system issues an appropriate message, such as "Most students take CS350 and CS360 together find the workload to be quite high". The design of SAM is such that hard constraints $T_1, \ldots, T_k$, are hard-coded into the interface, so they simply never arise as messages. The most important W_i's could cause an interruption, i.e. SAM takes control of the application and pops up a window forcing the user to focus on the warning. This is a somewhat drastic action appropriate only for the most important cases. For the less important warnings, SAM could issue them as back-channel utterances, for example a window containing the currently pending warnings could be kept on screen, and the user could read it whenever they want. By default, a good rule of thumb appears to be that interruptions should not occur too often because forcing the user to read a particular message can become irritating and counter-productive if done indiscriminately. Interruptions in SAM should be for important reasons: one could imagine a case where, in the past, every student who has taken a set of three particular courses has had to drop one due to the workload, so this might be something worth interrupting for, to make sure the student is aware of what they will be getting into if they take those three courses.

While "nature" does not play a role in the implementation of SAM, disruptions (i.e. nature taking the floor) have a natural interpretation in GUI environments that allow multi-processing. For example, one can imagine a window interface such that a new window pops up and takes control of the screen whenever a new piece of email arrives. Email can arrive at any time, so it is not something that the interface can be responsible for; hence, it is not an interruption. Such a system can let the user determine how

disruptions are handled. For instance, instead of popping up a window, a graphical flag might be raised, or the computer might make a beep, or it could do nothing at all.[15] Unexpected actions from other interruptions would also be categorized as disruptions.

Thinking of SAM in terms of the three-step turn-taking model, identifying concepts such as turn-ending signals, interruptions, and disruptions, provides a higher level of abstraction. It is possible, for example, to keep statistics on the occurrences and co-occurrences of these events, providing information which could prove valuable to an algorithm that tries to learn what constitutes conversational signals in a particular domain. Originally, SAM was conceived as a relatively simple graphically-oriented way of creating a student timetable as part of a system like that described in Figure 6.3. The intuitive idea was to roughly mimic on-screen how a student would create a schedule by hand. The formal description of scheduling invariant was derived when considering what the motivation step for SAM would be. Thus, part of the value of the turn-taking perspective is that it provides a systematic and consistent way of looking at interactive applications, which, as in this case, can sometimes lead one to make non-obvious generalizations. Furthermore, we have seen that thinking about the distinction between interruptions and disruptions, and utterances and back-channel utterances, can lead to design decisions affecting the look and operation of the interface. The value of the turn-taking theory for an application such as this is that it automatically provides a set of questions (related to turn-taking) that one can ask about their application, e.g. "What are interruptions?", "What are disruptions?", "What are turn-yielding signals", etc. While the results might lead to additions that one would add anyways, that could potentially be in an ad-hoc fashion, and so by mapping the system into the turn-taking model, one is naturally lead to these questions in a systematic way.

---

[15] This last case is behaviourally indistinguishable from how SAM currently acts since SAM ignores everything not part of the scheduling application.

Chapter 7

Related Work

7.1 Introduction

This chapter describes a number of pieces of related work. There is relatively little computational work directly concerned with turn-taking. Most AI research on dialog systems has focussed on higher level actions and relations, and turn-taking is usually either hard-coded into the system (e.g. this is the case with most typed-text systems, that require the user to give an explicit turn-ending signal by pressing the return key), or treated as an obvious and relatively unimportant concern.

7.1.1 Natural Language Generation

Natural language generation is the problem of describing a knowledge base in natural language. The problems in natural language generation range from selecting single words to generating multi-paragraph, multi-media texts (Bateman and Hovy 1992). Selecting the right word requires a fine understanding of the meaning and connotations of words; even a matter as seemingly mundane as deciding whether to use the word "a" or "the" is non-trivial (Knight and Chander 1994). Above the level of words, generation systems

not only need the linguistic knowledge necessary to generate grammatical sentences, but since there are typically many different ways to say essentially the same thing, they also require extra knowledge that enables them to choose from the varying possibilities. Such knowledge might include facts about the context or the domain, or knowledge about the reader/listener. For example, a description of how a phone works should be different if the reader is a child than if the reader is an engineer (e.g. see Paris 1993).

Generating multi-sentence text/dialog presents a new set of challenges. Knowing how to write good sentences does not necessarily mean one knows how to, or is able, to write good multi-sentence text. As discussed by Hovy (1993), language generation above the sentence level usually uses some form of textual relation to relate segments of the text, and patterns or rules that constitute good orderings are supplied, so the system can search for a realization that will fit the specifications. McKeown (1985) is an influential example of this approach, which has since been generalized, for example, by theories such as RST (Mann and Thompson 1988). Determining the best set of textual relations, and the right way to structure them, is a difficult task that presents many research challenges (e.g., see Moore and Pollack 1992, Hovy 1993).

Most work on generating natural language has taken a written-discourse perspective, although Section 7.2 discusses some interesting work on real-time language production. Thus there is no concern for resource constraints, and it is the structure of some knowledge base that directs the decisions of the generator. In contrast, the generation of stalling phrases discussed in Chapter 5 is caused by a running algorithm, and not some knowledge structure. This is a reflection of the idea that hesitations give the listener knowledge about the state of the generator's processing, and for written language this is not necessary since the writer has essentially all the time they need to write what they want.

If the generation of written-discourse can be thought of as describing a particular knowledge base, then generating spoken-discourse can be thought of as describing a

knowledge base *and* a running process. We need to include the running process to give an explanation for various speech-like phenomena, like hesitations and self-repairs (Section 7.2), which there is no obvious way to account for from a knowledge base alone.

7.2 Carletta et al.

Carletta et al. (1993b) outline an architecture for a time-constrained natural language generator, based on anytime algorithms. They are interested in modelling human language generation, and delineate explicit self-repair and hesitation modules. In particular, they want to be able to simulate *hesitations* and *spontaneous self-repairs* (Carletta, Caley, and Isard 1993a), which are both caused by time-pressure in human-human conversation. Hesitations occur when a speaker needs more time to think of what to say, and these gaps are often filled with stalling phrases such as *ermm*, or *you know*. They also consider hesitations manifesting themselves as stretched words, and repeated or redundant phrases. Spontaneous self-repairs happen when a speaker over-eagerly commits to an utterance, and must then go back and repair it; for example: "Go left, sorry, I mean right at that place with all the churches — Holy Corner". Carletta et al.'s goal is to develop a real-time human speech simulator, and so they are interested in both hesitations and spontaneous self-repairs.

This contrasts with the research presented in Chapter 5.1, which partly covers hesitations, but does not include spontaneous self-repairs. The reason for this is due in part to the general research strategy of this thesis, which is to start from the written-discourse perspective and evolve towards a spoken-discourse system. A simplifying assumption of written-discourse systems is that they are not time-pressured, nor are they expected to exhibit any kind of real-time performance. The stalls generated in Chapter 5.1 are caused by the running of a process, not the structure of a knowledge base, which is the usual

motivation for generation in a written-discourse system. Even a system that does not run in real-time can still generate stalls because they can give useful information about the state of the system to the listeners.

Spontaneous self-repairs are a feature of human language production, but they are not a necessary feature of language production in the abstract. We can dispense with the need for spontaneous self-repairs by simply never allowing the system to commit too quickly to generating an utterance. In any case where it is not quite sure what to say, the system can act conservatively and think a little more, possibly uttering a stalling phrase while doing this. In an application-oriented system like *SG*, it is not obvious that there is *any* situation when a mistaken and then immediately repaired utterance is preferable to an utterance that takes a little longer to generate, or contains some kind of stall. The actual use of spontaneous self-repairs in practice may be more useful in spoken dialog systems, where naturalness or variety of diction might be a reason to purposely allow for some self-repairs instead of all hesitations.

In application-oriented computer systems, particularly those that interact with people, it is difficult to justify why one would prefer a spontaneous self-repair over a hesitation. If a system realizes it is not certain about the correctness of an utterance, then it should hesitate until it is sure. If the system takes a chance and utters incorrect information that is repaired, e.g. *turn left — I mean right*, then there is always the possibility that the listener missed or mis-heard this repair. Uttering something that is potentially incorrect injects extra uncertainty in the dialog, uncertainty that can be avoided if a hesitation is used instead. [1] One possible argument in favour of self-repairs in a spoken dialog system is that they might make the system seem more natural to humans, which is generally a

[1] It may happen that a language generation algorithm is so complex or clever that such self-repairs emerge naturally in its processing without being programmed explicitly to choose between self-repairs and hesitations, but, however they arise, they might still add unnecessary uncertainty to the interaction.

good thing. However, if such naturalness caused too many problems, then it should likely be sacrificed in a less human-like, but safer, style of speaking.

Even though spontaneous self-repairs are not considered directly in this thesis, it is worth pointing out that the local search algorithms of Section 4.3.2 do not in principle disallow the possibility of self-repairs. For example, a system could be in the process of solving an instance of the next-action problem (Section 4.4), and then for some reason decides to act before it has definitely chosen the correct first action. It begins to execute the selected action, while simultaneously continuing to solve the next-action problem. It then discovers that it has selected the wrong action, so execution of the first selected action is terminated, accompanied with an appropriate repair-phrase such as *oops* or *no, sorry*, and then execution of the correct action begins.

7.3 Traum and Hinkelman

Traum and Hinkelman (1992) present a general model of *conversation acts* that situates turn-taking at the lowest level of a hierarchy of four different kinds of conversational actions. Their model of turn-taking shares many similarities with the logical theory presented in Chapter 5, although their theory is mainly descriptive and is not based on an explicit model of agents and a shared floor object. Their basic turn-taking actions are defined in terms of utterances and include *release-turn*, *assign-turn*, *keep-turn*, and *take-turn*. For example, they treat any instance of starting to talk as an attempted *take-turn* action, so they must model back-channel utterances as a *take-turn* followed quickly by a *release-turn*. They treat floor battles as a conflict between a *keep-turn* and *release-turn* action, and such a battle is won when one conversant stops speaking. They do not explicitly model offers or requests for the floor, although they do mention the possibility of a *pass-up-turn* action. We believe our model is more flexible since it separates turn-

taking actions and utterances, which allows both for utterances that do not affect the turn, and for the floor to be affected by non-utterances, e.g. disruptions by N.

Traum and Hinkelman use the idea of *local initiative*, which they claim means essentially what Walker and Whittaker (1990; Section 7.10) mean by *control*. Traum and Hinkelman write:

> *Local initiative* can be glossed as providing the answer to who has the most recent discourse obligation — who is expected to speak next according to the default plans for the simplest satisfaction of conventional goals. For example, a question or request will produce a discourse obligation on the other party to respond to the request (either by satisfying it, accepting it as responsibility to be performed later, or denying it).

They go on to state that local initiative is sometimes important in distinguishing keep-turn actions from release-turn actions, so clearly who has the turn and who has the local initiative are not the same thing in their model; this distinction is not as clearly made in Walker and Whittaker (1990; Section 7.10). We note that Collagen, discussed in Section 7.8, also uses a notion of local initiative.

The rest of their paper builds further levels of complexity on top of the basic turn-taking acts. *Grounding acts* are the level above turn-taking acts, and refer to coherent units of discourse; this includes actions such as initiating or continuing or canceling a discourse unit, acknowledging understanding of a previous discourse unit, repairing the content of the current discourse unit, or requesting an acknowledgment or a repair. This level is similar to the artificial meaning negotiation language described by Sidner in Section 7.9. *Core speech acts* are built on top of the grounding acts, and capture essentially the level of speech acts commonly used in computational discourse research (see Section 2.3). Finally, *argumentation acts* are composed of core speech acts, and are

meant to capture adjacency pairs (e.g. question/answer pairs) and higher-level rhetorical relations such as those described by Mann and Thompson (1988).

Traum and Hinkelman's goal is to develop a "grand theory" of conversation, and so they do not consider other possible applications of turn-taking, nor do they consider turn-taking outside of the conversational framework.

7.4 Cawsey's EDGE System

Cawsey (1992) discusses the theory and implementation of her EDGE dialog system, which is similar in spirit to Carberry's TRACK system (Section 2.3.2). While Carberry was concerned with generating intelligent advising dialogs (e.g. helping a student pick a schedule of courses), Cawsey focuses on the problem of generating good explanations (see also, for example, Paris 1993; Moore 1995). EDGE can handle cases where, unexpectedly, the user asks for an explanation of a concept, and it calls the resulting dialogs "clarification subdialogs", and treats these as interruptions of the main dialog (note that these clarification subdialogs are not quite the same as the similarly-named subdialogs in the CvBS system of Section 7.5). EDGE places special opening and closing utterances around such an interruption, for example:

A: Well, the light intensity is high so the resistance is low. Anyway ...

(This interruption occurs in a dialog about fixing an electronic circuit.) The boxed words represent the interruption opening and closing, and correspond to the "push" and "pop" operations of creating a new discourse segment. For short interruptions, using *anyway* as a closing works well for EDGE, but for longer interruptions, a longer closing might be necessary to remind the user of where the conversation has been, e.g. *Anyway, we were in the middle of explaining how the light detector works.*.

EDGE does not explicitly deal with turn-taking, although it does cite influences form the field of conversational analysis, which includes research such as Oreström (1983) and Sacks et al. (1978). Like most other dialog systems, it assumes the user issues an explicit turn-ending signal (e.g. pressing the return key). Thus, only certain kinds of semantic interruptions — and no mid-turn interruptions — can be dealt with by EDGE.

7.5 The CvBS System

Cohen et al. (1994) and van Beek et al. (1993) present a plan-critiquing approach to clarification dialogs in cooperative response generation systems; we refer to their system as the CvBS system. They work within a plan-recognition framework similar to that described in Section 2.3.2, and relax the common assumption made by such systems that the plan recognizer can uniquely determine what single plan the user is following. For example, in this exchange the user could be following any of a number of different plans:

Dialog 7.1

User: Can I drop numerical analysis?

System: Yes, but you will still fail the course since your mark will be recorded as withdrawal while failing.

Here, the system is giving more than just a simple yes/no answer to the user's surface question in an effort to be cooperative. However, although it has inferred that the user wants to drop numerical analysis in order to avoid failing it, that might not actually be the case: the student may want to drop the course due to a scheduling conflict, or because they find the material uninteresting. Van Beek et al. show how a clarification dialog can be used to accurately and efficiently determine the user's plan.

A key idea in their work is determining when ambiguity is relevant. If the system believes that, say, the user is following one of three possible plans, then, depending on the information sought by the user, it might not be necessary to determine exactly which plan the user is following. For instance, suppose the user asserts *I'm making marinara sauce*, and the system determines the user must be following one of these plans: "make fettucini marinara", "make spaghetti marinara", or "make chicken marinara". If the user then asks *Is a red wine a good choice?*, then there is no need to determine which particular dish the user is making since in any of the cases a red wine is a good choice. In other situations, though, ambiguity might matter; for example, if the system knows there is a vegetarian guest attending the dinner, and the user asserts *I am making marinara sauce*, then the systems needs to make sure the user is not following the faulty plan of making chicken marinara (a meat dish).

Figure 7.1 shows the top-level algorithm for how a response is generated in a cooperative dialog setting. The *Clarify-Ambiguity*() procedure works essentially by initiating a yes/no clarification dialog with the user whenever an ambiguity is detected. The plans in S are partitioned according to fault-type, and ambiguity matters only when there is more than one kind of fault-type to consider.[2] The goal is for the system to ask as few and as short questions as possible about the user in order to narrow the set of possible plans down to a single fault-class (which might contain many plans). Cohen et al. (1994) and van Beek et al. (1993) discuss a number of different strategies for managing these clarification dialogs. Their questioning strategies depend on the hierarchical structure of the plan library representation they are using, and they generally ask questions in a top-down fashion assisted by heuristics for choosing the next topic.

[2]There will usually be many more plans than plan-faults to consider. It is conceivable that there could only be one plan per fault class in some situations, and it that case all ambiguities would be relevant.

```
procedure Generate-Response(Query)
  if Query is not possible then
    tell user why Query fails
  else
    Make-Response(Query)
  end if
end Generate-Response

procedure Make-Response(Query)
  S←perform plan recognition on Query
  S←critique (and annotate) each plan in S
  if ambiguity in S matters then
    S←Clarify-Ambiguity(S)
  end if
  if plan S is faultless then
    tell user that Query is advisable
  else
    point out S's fault to user, and give reason/explanation for fault
  end if
end Make-Response
```

Figure 7.1: Van Beek et al. (1993) algorithm for response generation.

Like most plan-recognition based systems, the CvBS system does not concern itself with issues such as when to speak. As can be seen in Figure 7.1, the decision of when to initiate a clarification dialog is hard-coded into the algorithm and clarification is always begun as soon as it is recognized, continuing until the system is satisfied that the ambiguity has been resolved. Absent is any possibility for the user to jump out of a clarification dialog, in the event that they see what the system is getting at and are ready to offer up the desired information to the system right away, or if the user for some reason wants to escape from that particular subdialog. The CvBS clarification dialogs are always the system asking the user a series of yes/no questions, and no deeper subdialogs are permitted. Since the user can only answer yes or no, there is no reason to do plan recognition

on these utterances in the same way as done with top-level user utterances, although problems related to consistency may arise, e.g. the user mistakenly says *yes* when they meant *no*, or the user gives answers that are inconsistent with the system's beliefs about the user. Ideally, if something goes wrong during the clarification subdialog, then an appropriate repair subdialog should be initiated.

For the CvBS system to benefit from the turn-taking theory, it would first be necessary to enrich the clarification dialogs by allowing the user to respond with more than just yes or no to the system's questions. Currently, each yes/no response from the user corresponds to one bit of information that helps the system narrow-down the set of possible faults it is considering. However, the student may sometimes give responses that digress from the clarification without significantly affecting its outcome. For example:

Dialog 7.2

U: Can I take Cs100?

S: Are you a CS Major?

U: What is a CS Major?

A question like *What is a CS major?* can be handled independently of the yes/no responses, and in this case would probably amount to a one-turn description of "CS Major" followed by a return to the main clarification subdialog. Any kind of question that does not affect how the fault-classes are pruned could be handled this way.

More interesting are responses that influence or depend upon the reasoning process CvBS uses to generate clarification questions. For example, the user might sometimes provide more than a single bit of information:

Dialog 7.3

U: Can I take CS100?

S: Are you a CS Major?

U: I am a double major in CS and Physics.

The user has indirectly answered the question "yes", but they have also provided more information that could help the system; later in the clarification subdialog, there is no need for the system to ask questions like *Are you a physics major?* or *Are you a science major?*, because they already know what the answer will be from this response. A system's response might also depend on how the subdialog can continue:

Dialog 7.4

U: Can I take CS100?

S: Are you a CS Major?

U: I'm not sure yet.

From this the system can infer a "no" response, because the student is not currently a CS major. But a more helpful answer would explain how being a CS major affects CS100 eligibility. The system might respond:

Dialog 7.5

S: If you are a CS major, then you definitely cannot take CS100 for credit. But if you are not a CS major, then you might be able to take it for credit.

The second sentence says "might" because at this point there is still some relevant ambiguity, i.e. there are other circumstances that disallow the student from taking CS100, and the system would need to ask more clarification questions if the student was not a CS major.

Once it can handle more sophisticated utterances from the user, the CvBS system could then be extended to recognize turn-ending and turn-yielding signals. This would allow for the possibility of mid-turn interruptions, and, if the CvBS algorithms were re-implemented to be interruptible, or at least aware of their processing state, extensions like those presented in Section 5.4 could then be implemented.

7.6 Smith et al.

Smith et al. (1995) describe a PROLOG-based dialog system which is remarkable in a number of ways. Unlike other systems, it does not rely on a plan-based approach to dialog, but uses a novel architecture based on theorem-proving:

> The central mechanism of our architecture is a Prolog-style theorem-proving system. The goal of the system is stated in a Prolog-style goal and rules are involved to "prove the theorem" or "achieve the goal" in the normal top-down fashion. If the proof succeeds using internally available knowledge, the dialog terminates without any interaction from the user. Thus it completes a dialog of length zero. More typically, however, the proof fails, and the system finds itself in need of more information before it can proceed. In this case, it looks for so-called "missing axioms," which would help complete the proof, and it engages in a dialog to try to acquire them. (Smith, Hipp, and Biermann 1995)

Smith et al. built an a PROLOG simulator that can be arbitrarily interrupted, so the structure of their dialogs was not limited to the normal top-down order of Prolog; indeed,

their Prolog simulator is able to maintain partially completed proofs and jump among them as required by the progression of the dialog. The missing-axiom theory corresponds to the motivation step of the turn-taking model, and the theory sketched in Section 3.4 can be seen as an extension and partial elaboration of the missing-axiom theory. For instance, the discussion in Section 3.4 of what kinds of facts a person can be expected to know would have direct relevance to a system that works by following the missing-axiom theory. Also, their use of an interruptible PROLOG simulator addresses a major concern in this thesis, that dialog systems allowing for the most general kind of turn-taking (which includes interruptions) need to use algorithms that can tolerate interruptions caused by environmental resource limits.

Smith et al.'s system also incorporates a user model, allows for variable initiative settings (see Figure 7.2), and has been tested on a speech recognition/vocalization system that helps students fix electronic circuits. Their definition of initiative is interesting in part because it is practical enough to allow empirical testing of different levels of initiative, and it also clearly distinguishes itself from turn-taking.

7.7 Guinn's Model of Initiative

Guinn (1993) presents a model of dialogue initiative for collaborative discourse. Guinn's model is based on the missing axiom theory (Section 7.6), and he assumes two agents are collaborating on solving one problem. The agents both try to solve the problem on their own and then make requests of each other if they cannot solve subgoals alone. When an agent receives a request for help, it must decide if it should deal with the request or continue solving its own problems. An agent's decision is based partly on the (user) model it has of the other agent.

The model of motivation outlined in Section 3.4.1 shares many similarities with

Directive Unless the user explicitly needs some type of clarification, the computer will select its response solely according to its next goal for the task. If the user expresses need for clarification about the previous goal, this must be addressed first. No interruptions to other subdialogs are allowed.

Suggestive The computer will again select its response according to its next goal for the task, but it will allow minor interruptions to subdialogs about closely related goals. As before, user requests for clarification of the previous goal have priority.

Declarative The user has dialog control. Consequently, the user can interrupt to any desired subdialog at any time, but the computer is free to mention relevant, thought not required, facts as a response to the user's statements.

Passive The user has complete dialog control. Consequently, the computer will passively acknowledge user statements. It will provide information only as a direct response to a user question.

Figure 7.2: Four modes of variable initiative in the voice-dialog system implemented in Smith et al. (1995). The language generation module of their system has an "assertiveness" switch that corresponds to the level of initiative, so the system will have a different style of speaking depending upon the initiative mode.

Guinn's work. Both exploit the idea of asking another agent when one's private problem-solving process fails. Guinn's model is implemented and he reports results on a number of different strategies for changing the initiative. A major result of Guinn's work is to show that different initiative strategies can result in significantly shorter/longer dialogs. In particular, with the strategy Guinn calls *continuous selection*, the problem-space can have an exponentially large number of nodes pruned away.

Guinn uses a very simple user model consisting of a list of facts about paths in the problem space. It does not appear that Guinn's model allows for inaccurate user models, which in practice would likely lessen the effectiveness of his algorithms.

7.8 Rich and Sidner

Rich and Sidner (1998) describe Collagen, a toolkit for building collaborative interface agents. Collagen is domain-independent toolkit based on the SharedPlan model of discourse (Grosz and Sidner 1990), and provides a kind of mixed-initiative interaction and a reviewable dialog history based on Sidner's artificial negotiation language (Section 7.9). For Rich and Sidner, a *collaborative interface agent* is an agent that assists a user to perform some task on a shared artifact; for example, two mechanics fixing a car, or a pair of students making a poster.[3] It is assumed that the interface agent can communicate with the user, and, independently, with the shared artifact (typically some application program).

They demonstrate Collagen using a sample air travel application. The user is presented with an active map of the USA surrounded by buttons, sliders, message boxes, and windows, which allow them to build an itinerary of flights between cities. A constraint-checker[4] is built into the application, so the user cannot input inconsistent information, and the typical problem that this application solves involves visiting multiple cities with time constraints, e.g.

> "You want to visit Dallas, Denver, and San Francisco. You start in Boston, and want to leave on Wednesday night if possible, but will settle for Tuesday morning. You prefer American Airlines, since you have frequent-flier points with them that you can use. You must be home by 5pm on Friday."

Collagen augments this base application by adding a collaborative interface agent

[3]Presumably this would *not* include purely information-giving systems, even highly cooperative and intelligent ones. For example, an intelligent kiosk at the train station might dispense facts about the train schedule to anyone who asks, but since there is no shared artifact, this would not be an example of a collaborative interface agent in Rich and Sidner's terms.

[4]SAM, discussed in Section 6.5 also uses a constraint-checker, which simply determines whether or not a set of value satisfies a set of constraints.

that assists the user in creating an itinerary. The most visible change is the addition of two "home windows" that sit on top of the application screen, one for the user and one for the interface agent. The user window contains a picture of the user, and the agent window has a picture of the agent, whose eyes blink periodically to indicate its process is still running. The interface agent uses its window to display text messages, and also, sometimes, to unobtrusively get the user's attention by waving its hand when it has something to say. It is important to note that Collagen provides neither the application acting as the shared artifact, nor the implementation of the assisting agent (although a very simple default agent is provided); it provides only an interface and set of conventions for the actual interface agent.

A major feature of Collagen is its use of an interaction history, which is a recording of all the actions done by the user and the system. This history is a text listing of the actions performed, available at any time to the user and indented according to the discourse structure of the interaction. The user can manipulate contiguous chunks of this history, and perform actions such as replaying or undoing those parts of the interaction. While the internal representation is an artificial negotiation language (Section 7.9), the dialog history presented to the user is a simple English-like gloss of this representation, which is somewhat easier to read.

Collagen distinguishes between global and local conversational initiative. Local initiative phenomena include:

- turn taking and conversational control (who gets to speak next);
- interruptions;
- grounding (how the speaker indicates she has heard and understood the previous speaker's utterance).

They describe global initiative as "... [focusing] on the conversants' problem-solving level

and on choosing the appropriate speech acts or discourse plan operators." While Rich and Sidner believe that Collagen provides good support for global initiative, it does not currently provide any direct support for it since they believe it is still a major research issue. Collagen does provide direct support for local initiative. For instance, the user can relinquish control to the agent by clicking on "ok" at some time other than during answering a yes/no question. And as mentioned above, the agent can passively attempt to get the user to relinquish the turn by waving its hand.

The turn-taking theory presented in this thesis could be one of the tools of a Collagen-like toolkit. The turn-taking logic of Section 5.3 provides both a formal language for describing turns, and turn-taking protocols discussed in Section 5.3.9 could be provided as starting points to giving more meaning to the structure of the dialog history. For example, interruptions and disruptions could be marked, which could be of benefit to the user. One difference between Collagen and the work in this thesis is that Collagen, while a real-time interactive system, does not provide any direct support or advice for using algorithms that can support interruptions (e.g. anytime algorithms) or are appropriately sensitive to resource bounds. The internal workings of the interface agent are left up to the toolkit user, and so they can use any kind of algorithm they want, which means they can use resource-sensitive ones, but the rest of the tools do not necessarily know about such details.

7.9 Sidner's Negotiation Language

Sidner (1994) presents an artificial language for negotiation in discourse. It allows discourse segments to be annotated as offers, proposals, rejects, acknowledgments, etc. This language shares some surface similarities with the turn-taking logic of Section 5.3, however Sidner is concerned with detailing a language for modelling the structure of discourse

(similar to Walker's artificial language, mentioned in Section 7.11), while the turn-taking logic is a formal description of the conversational floor. Sidner models acknowledgments explicitly, but we do not, instead assuming that all agents correctly know who has the floor at all times (except during floor battles). For arbitrary conversational actions this is not a reasonable assumption, because agents can easily be mistaken about whether or not they have been understood, and so acknowledgments can provide useful feedback. For the conversational floor, however, mistaken beliefs are rarely an issue, particularly in two party dialog where mistakes about the floor will, for example, cause both conversants to simultaneously speak or not speak.

7.10 Whittaker et al.

Building on earlier work by Whittaker and Stenton (1988), Walker and Whittaker (1990) investigate *initiative* and *control* in a dialog setting. While Walker and Whittaker do not give precise definitions of initiative and control, they do introduce them as follows:

> Conversation between two people has a number of characteristics that have yet to be modeled adequately in human-computer dialog. Conversation is BIDIRECTIONAL; there is a two way flow of information between participants. Information is exchanged by MIXED-INITIATIVE. Each participant will, on occasion, take the conversational lead. Conversational partners not only respond to what others say, but feel free to volunteer information that is not requested and sometimes ask questions of their own [Nic76]. As INITIATIVE passes back and forth between the participants, we say that CONTROL over the conversation gets transferred from one discourse participant to another.

While it is unclear what to make of the sentence "Information is exchanged by MIXED-INITIATIVE", it appears that Walker and Whittaker define initiative to be something

Utterance Type	Controller
Assertion	Speaker, unless response to a Question
Command	Speaker
Question	Speaker, unless response to a Question or a Command
Prompt	Hearer

Table 7.1: Control transfer rules from Walker and Whittaker (1990).

similar to the conversational floor, a single shared object that conversants pass back and forth between themselves. Also, presumably the participant with the initiative is the one in control of the conversation, although the above quote does not enforce that interpretation.

Walker and Whittaker (1990) define their control transfer rules on utterance types, as done in Whittaker and Stenton (1988). They list four kinds of utterances:

Assertions declarative utterances used to state facts;

Commands utterances intended to instigate action;

Questions utterances intended to elicit information;

Prompts utterances which do not express propositional content, such as *Yeah*, *Okay*, *Uh-huh* ....

Dialogs are segmented according to these utterance types, and the rules in Table 7.1 define who is in control of each segment.

Walker and Whittaker also generalize a set of interruption rules first given in Whittaker and Stenton (1988), which are listed in Figure 7.3. These are very general rules, and terms such as "relevant" and "ambiguous" are not defined. For example, rule A2 says that a listener should interrupt when it hears relevant but ambiguous information. Suppose Madge is getting directions from Trygve about how to get to a gas station:

Dialog 7.6

Trygve: Go down this road and you'll see it at the corner of the first intersection you come to.

Since intersections usually have more than one corner, it would seem that Trygve's assertion is both relevant and ambiguous, so Madge should interrupt. Yet, this is not an important ambiguity since Madge should have no trouble finding the gas station, whichever corner it is on at the intersection. This particular idea is explored further in the CvBS system (Section 7.5), which only begins a clarification dialog when there is *relevant ambiguity.*

Another shortcoming of the rules in Figure 7.3 is that they correspond only to the motivation step of our turn-taking model (Chapter 3) and do not tell an agent *when* to actually interrupt. For example, suppose Trygve gives these directions:

Dialog 7.7

Trygve: Go down this street, and turn left.

When Trygve speaks the word "turn", Madge could treat everything he has said up to that point as a relevant but ambiguous utterance, since he has not (yet) specified which direction she should turn. Clearly, there needs to be some notion of when to interrupt, otherwise Madge might unhelpfully and annoyingly interrupt just after the word "turn".

7.11 Walker and Resource Bounds

Walker (1993, 1994) describes research that treats conversation as a resource-bounded activity. The "efficiency" of a dialog is typically measured by its length, but Walker points out that cognitive effort should also be taken into account, especially for resource-limited agents such as humans. Most problem-solving systems assume that they will have

Information Quality The listener must believe that the information the speaker has provided is true, unambiguous, and relevant to the mutual goal. This corresponds to the two rules:

(A1) Truth If the listener believes a fact P and believes that fact to be relevant, and either believes that the speaker believes not P or that the speaker does not know P then interrupt;

(A2) Ambiguity If the listener believes that the speaker's assertion is relevant but ambiguous then interrupt.

Plan Quality The listener must believe that the action proposed by the speaker is part of an adequate plan to achieve the mutual goal and the action must also be comprehensible to the listener. The two rules to express this are:

(B1) Effectiveness If the listener believes P and either believes that P presents an obstacle to the proposed plan or believes that P is part of the proposed plan that has already been satisfied, then interrupt;

(B2) Ambiguity If the listener believes that an assertion about a proposed plan is ambiguous, then interrupt.

Figure 7.3: Interruption rules from (Walker and Whittaker 1990).

enough memory to hold all the necessary constraints or predicates that might come into play, and concentrate mostly on minimizing time.

Walker deals with *informationally redundant utterances*, or IRUs, which appear in many human-human conversations. Figure 7.4 gives an example of a naturally occurring IRU. For agents with no resource bounds there is little reason to repeat an utterance that does nothing but add already established information. However, as Walker demonstrates through analysis and computer simulation, IRUs prove very helpful to agents who have only a limited working memory.

Walker uses a cognitively plausible model of limited memory called the Attention/Working Memory, or AWM, model. Walker (1994) describes it as follows:

H: Right. The maximum amount of credit that you will be able to get will be 400 *that they will be able to get will be 400 dollars on their tax return*

C: *400 dollars for the whole year?*

H: *Yeah it'll be* 20%

C: *um hm*

H: Now if indeed they pay the $2000 to your wife, that's great.

C: um hm

H: So we have 400 dollars

C: Now as far as you are concerned, that could cost you more ...

Figure 7.4: An IRU from (Walker 1994). This is a dialog from he Harry Gross transcripts, recordings of a call-in radio show where callers ask Harry for financial advice. The boxed utterance is an IRU, and the italicized text marks the dialog that added the original belief to the context.

> AWM consists of a three dimensional space in which propositions acquired from perceiving the world are stored in chronological sequence according to the location of a moving memory pointer. The sequence of memory loci used for storage constitutes a random walk through memory with each loci a short distance from the previous one. If items are encountered multiple times, they are stored multiple times [Hintzman and Block, 1971].

Memory retrieval works in a spherical fashion, outward from the origin. Bounds are enforced by limiting the radius of the sphere.

Walker's experiments were carried out in the "Design-World" testbed, where agents must communicate and collaborate on arranging furniture. A simple artificial language was defined, and agents could perform the communicative acts of PROPOSE, ACCEPT,

REJECT, and SAY. Figure 7.5 gives a sample of the kind of dialog her system generated. Walker (1994) discusses these three different communicative strategies (Walker 1993 includes a number of other strategies):

All-Implicit this strategy has no IRUs, and no conversational acts, such as acceptances or openings, are implicit;

Close-Consequence IRUs are inserted at the close of a dialog;

Explicit-Warrant IRUs includes a WARRANT IRU for each proposal made.

The agents were set different tasks, all centered around coming up with a plan for designing the layout of a house. The standard task was scored by summing the valid steps of the resulting plan and subtracting the penalties due to invalid steps. More severe tasks were also tested: the "zero invalids" task requires that no step of the plan be invalid, and the "zero non-matching beliefs" task required that the agents agree on all their reasons (warrants) for performing a particular action.

By simulating agents running different communication strategies trying to solve different tasks, Walker's experiments produced a number of interesting results about trade-offs between processing time and memory size. For instance, the Explicit-Warrant strategy is a good choice for the zero non-matching beliefs task when agents have small limited memory. The reason is that retrieving a warrant from memory can be costly, so it is advantageous to state warrants with proposals, and thus save memory. At larger working memory sizes, the advantages of this strategy over the others becomes less pronounced. Essentially, while IRUs have some cost due to the overhead of speaking and repeating information, they can save memory space, and in the right circumstances, result in a net benefit. At other times, especially when there is more working memory, the communication overhead of the IRUs outweighs the benefits of the saved memory. Walker (1993)

BILL: *Then, let's put the green rug in the study.*
(propose agent-bill agent-kim option-43: put-act (agent-bill green rug room-1))

KIM: *Then, let's put the green lamp in the study.*
(propose agent-kim agent-bill option-61: put-act (agent-kim green lamp room-1))

BILL: *No, instead let's put the green couch in the study.*
(reject agent-bill agent-kim option-75: put-act (agent-bill green couch room-1))

Figure 7.5: Sample Design-World dialog from Walker (1994). Walker's system only dealt with the propositions shown below the text. The text is a gloss provided by Walker.

goes into much detail on the benefits of various communication strategies in different dialog situations.

In contrast to Walker, this thesis has been more concerned with time-constraints. This is partly because, unlike humans, computers do not have any similar short-term memory limitations. Walker's work could have applications to user modeling, because humans are limited by the working memory constraints she models, so a cooperative system should be aware of this, and, for example, generate appropriate IRUs to better accommodate users. Time-constraints apply more equally to both humans and computers, and so they are the more natural resource bounds to consider when talking about agents in general. It would be straightforward to limit the size of, for example, *SG*'s working memory, such that it could only hold a certain number of goals at once. This would not be of any use to *SG*, where working memory can be very large, and it would not be of much cognitive interest (in the way that Walker's work is) unless it were re-worked to instantiate a more cognitively plausible architecture. Interesting future work would be to build an agent that takes both time and Walker-style memory constraints into account.

Chapter 8

Conclusions

> I haven't been this happy since the end of World War Two.
>
> *Leonard Cohen, Waiting for the Miracle*

8.1 Thesis Summary

This thesis has presented a computational model of turn-taking in dialog. Most AI dialog systems focus exclusively on determining *what* to say, and leave the question of when to speak essentially untouched. In the context of turn-taking, when to speak is related to control of the conversational floor, and Chapter 5.1 presented a formal logical account of a turn-taking that allows one to speak precisely and unambiguously about turn-taking. Furthermore, this was presented as the third step of a general turn-taking architecture (Chapter 3). The initial motivation step is concerned with translating perceptions into goals/actions that can be reasoned about by the second step. The second step of the general model shows how local search algorithms can be used to select and order multiple goals/actions in a way that is sensitive to the time-bounds of the interaction; Chapter 6 presented the *SG* system, which demonstrates that the action-selection paradigm presented in Chapter 4 is a practical one.

8.1.1 Summary of Contributions

- Identification of turn-taking as a novel and basic perspective in multi-agent systems. Conversational turn-taking, while the original motivation for the general view, is just one specialization of the general turn-taking model. The abstract notion of turn-taking consists of two or more agents and an indivisible object they share among themselves. In conversation, this object is the abstract "floor". In soccer, it is the ball; in an intersection, it is the physical space taken up by the crossing of the roads (Section 5.3.4);

- The specification and elaboration of a general turn-taking architecture:

  Motivation Section 3.4 discusses the general framework for how a turn-taking agent might deal with the motivation step. There it is suggested that propositions be classified according to how "difficult" an agent expects coming to know a proposition will be. This information can then be used by the agent to help make decisions about whether or not it should try to satisfy a goal by thinking about it itself, or asking another agent. The motivation work presented in this thesis is a first step, and, as discussed in Section 8.2, there are a number of interesting and potentially fruitful directions in which it can be carried;

  Reasoning Chapter 4 introduced the idea of the next-action problem, and presented a CSP-based formalism for solving it in a resource-sensitive way. The next-action problem allows an agent to select among and order a large set of goals, and so is one approach to solving the general planning problem, most appropriate for highly uncertain or dynamic domains. By solving the next-action problem with local search algorithms, a dynamically changing environment can be handled in a natural way, and the anytime properties of the algorithms can

be exploited to monitor the amount of processing completed by the algorithm. This sensitivity to resources and the environment is necessary in a turn-taking system, since an agent could be required to act when it is not completely prepared to act. Results of empirical testing on a number of plausible local search algorithms were given in Section 4.6;

Execution Chapter 5 introduces a formal model of turn-taking that describes the problem from a third-party point of view. This provides a clear and concrete description of turn-taking, which clarifies the turn-structure of dialog, and also sheds light on the subtle concept of initiative. The logic is also used to develop a single-agent algorithm (in terms of a state/action table) for taking a turn in a dialog that is sensitive to the state of processing of the algorithm. The algorithm specifies when an agent should take a turn and introduces "stalling" phrases, which an agent uses to buy more thinking time.

- Chapter 6 uses the turn-taking theory to develop a list of questions that can help in the design and evaluation of interactive systems. Implementations of two systems that solve (essentially) the same course-advising problem in substantially different ways were presented: *SG* is a standard command-line dialog system, where the user types in commands/queries, and the system response with written replies (both natural language and menus of choices); SAM is a direct-manipulation interface where a user builds their schedule of courses by selecting courses from an on-screen list using a mouse. Both systems are then analyzed in terms of the general turn-taking questions (Figure 6.2), and a number of insights and ideas for improvement are suggested. We believe that these questions can be usefully applied to essentially any interactive multi-agent system that one wants to understand as a dialog system; the result can be insights into the structure and behaviour of the system which can

influence its design and future evolution;

- Another general contribution of this work is to have illustrated one path of evolution between written-discourse systems and spoken-language discourse systems. By starting with a standard written-discourse system and its attendant assumptions, the addition of turn-taking is a step towards a spoken-language system, since taking turn-taking seriously requires greater attention to resource limitations and the dynamic nature of real-time dialog. There are still steps to be taken, but turn-taking provides a good, solid first step in a process of refinement that will ultimately lead to spoken-language dialog systems based on consistent and systematic design principles inspired by human conversation.

8.2 Future Possibilities

I've seen the future, baby: it is murder. *Leonard Cohen, The Future*

This thesis presents a number of possibilities for future research. We list a number of ideas below:

- The initial motivation step described in Section 3.4 is the least developed of the three major parts of a turn-taking agent, and so expanding it would be useful. The motivational theories could be further elucidated, or other possibilities could be considered, e.g. the introduction of a general *ask-user* function to a computer language that can replace any function calls with a "call" to the user that returns the same type as the original function. For instance, an *ask-user* wrapper predicate could be added to PROLOG such that at any choice point, it retrieves all possible choices and presents the user with a menu of those choices to choose from. As discussed in Section 3.4, determining in general what sorts of operations are best

asked of the user is non-trivial, and this presents an interesting area for future research;

- The empirical study of the next-action problem in Section 4.6 was necessarily brief, and a larger and more systematic one would be of interest. Since the next-action problem and traveling salesman problem are relatively similar, almost any traveling salesman solution algorithm is a candidate for application to the next-action problem, and there are an immense number of traveling salesman algorithms to choose from (Reinelt 1994);

- The next-action problem allows for a kind of interruptible planning, which is useful in dynamic domains. As discussed in Section 2.3.2, plan recognition is the problem of inferring an agent's plans from observing their actions, and this is quite often useful in dialog. Plan recognition based on the next-action problem would consist of inferring an agent's penalty matrix by observing their actions. For example, if you believe an agent has a choice of performing (at least) action α or β, and the agent performs α first, then you can infer that the agent likely prefers α to β (although time-pressure could have caused it to act too soon and make a mistake). This would allow for interruptible anytime processing if one starts with a fully instantiated matrix and modifies its values as actions are observed. The initial matrix might be based on one's own penalty matrix for the domain;

- Turn-taking systems that interact, e.g. specify the relation between two floor-like objects that can each individually be described by the turn-taking logic. As suggested in Section 3.1, physical locations can be thought of as possessing objects, instead of the objects having locations; it is possible to treat locations as discrete objects that can contain at most one object at any one time. The turn-taking logic could then be applied to each location, allowing for objects to negotiate over

locations, i.e. make requests or offers to each other to have possession (i.e. be in) a location, etc. It is not obvious what the value of this way of looking at the world might be, although it is close in spirit to the idea of treating everything in the world as an agent (Shoham 1993);

- Resolving floor battles, or, more generally, resolving conflicting actions, was discussed only briefly in Section 5.3.8. More sophisticated conflict-resolution procedures could be developed, although resolving conflicts can be a very difficult task in general, especially for people. There are problems on the micro-level, i.e. how can two objects avoid being in the same place at the same time, and there are macro-level problems, such as preventing countries from going to war. While AI does not yet appear ready to deal with solving any macro-level conflicts, useful insight into micro-level conflict resolution might be gained by studying how human negotiators prevent and resolve conflicts;

- While the turn-taking logic of Chapter 5 allows for any number of agents to share the floor, only two-party interaction is considered, partly for simplicity and partly because it is sufficient for human-computer interaction. A natural question is to consider how turn-taking works among more than two agents. The turn-taking logic is a reasonable starting place, but it would likely need to be extended to account for phenomena that does not occur in the two-party case, such as listener communication. For example, someone might be speaking, while at the same time listeners may be passing non-disruptive signals among themselves that help to establish who will next get the floor. It is also possible for subinteractions to break out, where, for example, a pair of people in a big enough group might begin an interaction between themselves without interrupting the current speaker;

- A formal theory of initiative would clarify many issues, and the turn-taking theory of Chapter 5.1 provides a reasonable starting point for constructing a theory of initiative, e.g. one can start by treating initiative as an object that conversants can share, just like the conversational floor. The relationship between the floor and initiative would need to be made clear, and it is not obvious if initiative really is best modelled as a floor-like object;

- As pointed out in Section 3.3, the motivation step of the general turn-taking model also needs to be implemented using algorithms that are sensitive to the resource constraints of the interaction. In this step, the problem was skirted by dealing with simple rule-based systems (like SG) that directly map perceptions into goals in a very efficient way as compared to the reasoning step. If one wants to do more substantial processing here, such as parsing natural language or performing plan recognition, then the algorithms must be implemented such that their processing state can be monitored by the turn-execution module, just as it monitors the algorithm of step two. An interesting project would be to develop interruptible versions of standard dialog system components, such as parsers or plan recognizers, so that they can return a less then optimal solution if required to act, or give useful information about the state of its processing, e.g. estimate how long it will take to return a solution of a particular quality;

- Dialog systems are good candidates for intelligent user interfaces. However, significant user-testing is necessary to know what aspects of turn-taking (or indeed any kind of intelligent behaviour) users actually find useful. For instance, the turn-taking logic allows interruptions to be described, and one can imagine a user interface that occasionally interrupts the user. But interruptions are tricky to get right: interrupting the user at the wrong time might annoy them and cause more

trouble than if there had been no interruption at all. It would be interesting and useful to empirically study when, if ever, interruptions are more of a help than a hindrance

- The research strategy of this thesis, outlined in Section 1.2, is to start with written-language discourse and move it a step closer to spoken-language discourse by adding the concept of the conversational floor. Another way to go is to start with spoken language, and directly add an awareness of the conversational floor to that. That project would share some similarities with this thesis, but, since spoken language is a substantially different medium of communication than written language (Section 2.5), and so would form a major project on its own. Indeed, it would be valuable to develop a spoken-language system that is aware of turn-taking from the start.

That's it man, game over man, game over, man! Game over!

Private Hudson, Aliens

Appendix A

Generating a Penalty Matrix

A.1 Introduction

This appendix shows a way to automatically generate the penalty matrix for the next-action problem (Section 4.4) starting from a set of precedence graphs.

A.2 Creating a Penalty Matrix from a Precedence Relation

The penalty matrix directly encodes information about a precedence relation. If action a precedes action b, then this implies that a should be performed before b. The precedence relation can be thought of as a directed acyclic graph, called a *precedence graph*; Figure A.1 gives some example graphs and their corresponding matrix descriptions. A precedence graph should be acyclic because precedence is transitive[1] and so a cycle will enable a contradiction to be derived, i.e. that some action a both precedes, and is preceded by, some action b.

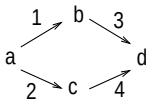
In what follows, we will show how a penalty matrix can be derived from a precedence

[1] I.e., if a precedes b and b precedes c, then a precedes c.

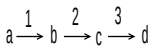
relation. We assume that this relation is acyclic, and the strength of the precedence between elements is represented with natural numbers. For a precedence graph with nodes $v_1, \ldots v_n$, we associate a cost matrix C such that entry C_{ij} represents the value (cost) on the edge going from v_i to v_j. These costs correspond to the penalty incurred for v_j coming before v_i in the next-action problem, although the penalty matrix itself is a slightly different form, being essentially the transpose of the cost matrix (Figure A.2).

We will assume that if a precedes b, then there is a penalty of 0 for a occurring before b (in the next-action problem), and some non-zero penalty for b occurring before a. The main problem is to determine the appropriate values for entries not corresponding to an edge in the precedence graph, i.e. to calculate the transitive closure of the relation. Consider the straight-line graph in the upper-right of Figure A.1. Since precedence is transitive, we can conclude that a should precede d, so the (a, d) position in its cost matrix should be 0 (similarly, a precedes c and b precedes d). The (d, a) cost-value is not as clear cut. If d and a occur out of order, then a penalty must be accrued. Since a and d are far apart, it seems reasonable to assign a penalty equal to the cost of the path from a to d, in this case 6 (i.e. the sum of the constituent edge costs). Now consider the diamond-shaped graph in the upper-left of Figure A.1. What should be the value of the (d, a) position? The problem here is that there are two paths from a to d: $P_1 = (a, b, d)$, and $P_2 = (a, c, d)$. The cost of path P_1 is 4, and P_2 has cost 6. In this case, we make the conservative judgement that the (d, a) value should be the *maximum* cost of a path from a to d, i.e. cost 6 due to path P_2.

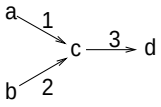
This problem is very similar to the all-pairs shortest path problem, which has a number of different good solutions (Cormen et al. 1990). The difference is that we want the *longest* paths, not the shortest. Figure A.2 gives an algorithm based on the Floyd-Warshall method, for calculating a complete description of a single transitive, acyclic relation (such as precedence).



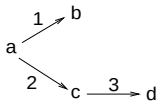
	a	b	c	d
a	·	1	2	$-\infty$
b	$-\infty$	·	$-\infty$	3
c	$-\infty$	$-\infty$	·	4
d	$-\infty$	$-\infty$	$-\infty$	·



	a	b	c	d
a	·	1	$-\infty$	$-\infty$
b	$-\infty$	·	2	$-\infty$
c	$-\infty$	$-\infty$	·	3
d	$-\infty$	$-\infty$	$-\infty$	·



	a	b	c	d
a	·	$-\infty$	1	$-\infty$
b	$-\infty$	·	2	$-\infty$
c	$-\infty$	$-\infty$	·	3
d	$-\infty$	$-\infty$	$-\infty$	·



	a	b	c	d
a	·	1	2	$-\infty$
b	$-\infty$	·	$-\infty$	$-\infty$
c	$-\infty$	$-\infty$	·	3
d	$-\infty$	$-\infty$	$-\infty$	·

Figure A.1: Sample precedence graphs. These are *not* penalty matrices, but instead *cost* matrices. The entries are set up to work with the Floyd-Warshall algorithm (Figure A.2). The positive edge values represent the penalty points assigned if the corresponding vertices do not appear in the given order, and $-\infty$ whenever a non-diagonal entry's value is unknown.

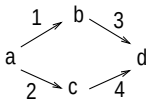
A.3 Multiple Precedence Relations

The result of applying the algorithm in Figure A.2 to a precedence graph is a complete penalty matrix for the relation. The actions of most interesting domains will be related in many ways, resulting in more than one set of (possibly contradictory) precedence relations (Section A.4). For example the network in Figure A.4 has two relations, precedence and decomposition. The decomposition relation implies a precedence ordering, so it can be translated directly to a precedence graph (Figure A.5). It is also possible to create a penalty matrix for multiple relations by summing the penalty matrices for each individual relation. So for Figure A.4, we would create a penalty matrix P_1 for the precedence relation, and a penalty matrix P_2 for the decomposition relation, and the overall penalty matrix is $P = P_1 + P_2$. The final penalty matrix for Figure A.4 is shown in Figure A.6. In this case, since the resulting penalty matrix is acyclic, it is possible to run the algorithm in Figure A.2 again, to calculate some of the intermediate values. This cannot be done

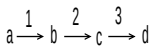
```
procedure FloydWarshall(G'[1...n, 1...n])
  G←G'
  for k← 1 to k do
    for i← 1 to i do
      for j← 1 to j do
        G(i,j)←max(G(i,j), G(i,k)+G(k,j))
      end for
    end for
  end for
  return G
end FloydWarshall

procedure
CreatePenaltyMatrix(G[1...n, 1...n])
  P''←FloydWarshall(G)
  P'←replace every −∞ in P'' by 0
  P←matrix transpose of P'
  return P
end CreatePenaltyMatrix
```

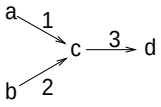
Figure A.2: Calculating a penalty matrix with the modified Floyd-Warshall algorithm. Conceptually, this algorithm finds the most costly path between each nodes i and j, and puts the cost in position (a, b) of matrix G. The regular Floyd-Warshall algorithm solves the all-pairs shortest path problem, i.e. given a graph with positive edge values, it returns a matrix where the (i, j)th entry is the length of the shortest path from node i to node j. In the above version, we essentially use "max" instead of "min". This requires two extra assumptions about the input: G must use negative infinities instead of positive infinities to represent unconnected nodes, and G must be an acyclic graph, otherwise one could go around and around a cycle and create an arbitrarily long path. The input matrix G should look like the matrices in Figure A.1; the results of applying *CreatePenaltyMatrix* to them are shown in Figure A.3.



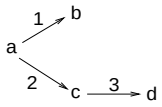
	a	b	c	d
a	·	0	0	0
b	1	·	0	0
c	2	0	·	0
d	6	3	4	·



	a	b	c	d
a	·	0	0	0
b	1	·	0	0
c	3	2	·	0
d	6	5	3	·



	a	b	c	d
a	·	0	0	0
b	0	·	0	0
c	1	2	·	0
d	4	5	3	·



	a	b	c	d
a	·	0	0	0
b	1	·	0	0
c	2	0	·	0
d	5	0	3	·

Figure A.3: Resulting penalty matrices from Figure A.1.

in general, since there is no guarantee that P is acyclic.

Essentially this method of creating a penalty matrix was used in the creation of the penalty matrix used in the *SG* system discussed in Section 6.4. In practice, hand-made adjustments were sometimes helpful, but this was straightforward and not difficult.

A.4 Other Relations

AI-style actions/goals are typically related in a number of ways. Two common relations that often appear in planning systems are precedence and decomposition. Precedence is the same as in Section A.2; for example, if α is "cook dinner" and β is "eat dinner", and since dinner should be cooked before being eaten, we stipulate that α precede β. Decomposition refers to how an action can be (hierarchically) composed of smaller actions. For example, the act of cooking dinner is composed of many smaller actions, such as turning on the stove, mixing ingredients, gathering cutlery, etc. As in Figure A.4, decomposition and precedence relations often go together in planning systems and other AI knowledge bases.

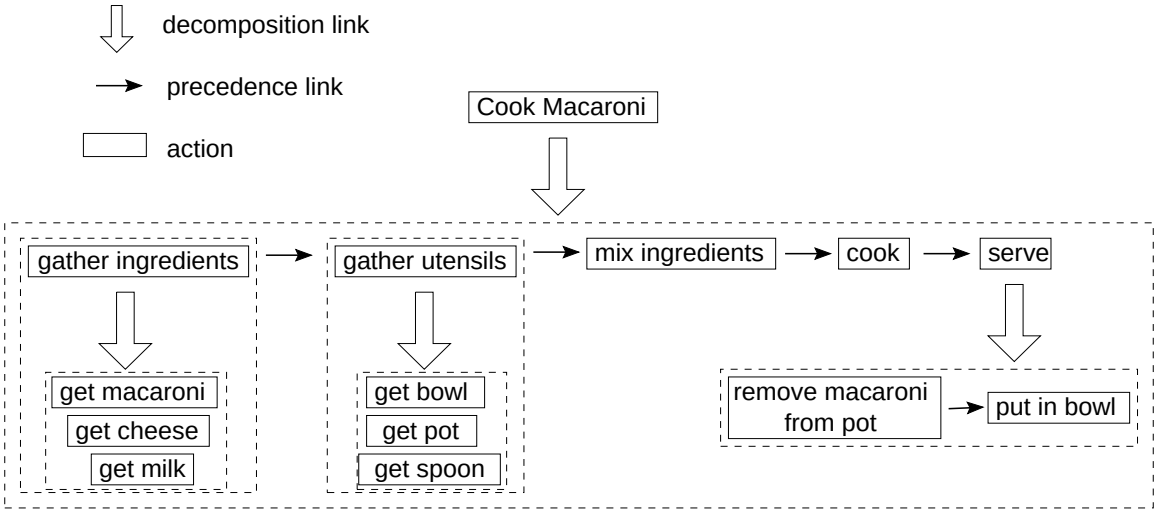


Figure A.4: How to eat like a king. This is a typical planning-style hierarchical decomposition network. For example, it does not matter in what order the ingredients are gathered in, but all the ingredients must be gathered before any utensils. Note that if an action α consists of subactions $\alpha_1, \ldots, \alpha_n$, and action β consists of subactions $\beta_1, \ldots, \beta_m$, and α precedes β, then that does *not* necessarily mean that all of actions $\alpha_1, \ldots, \alpha_n$ must precede $\beta_1, \ldots, \beta_m$. All that is required is that not all of $\beta_1, \ldots, \beta_m$ be finished before all of $\alpha_1, \ldots, \alpha_n$ are finished. If we do want to require that all of α's subactions be finished before any of β's subactions (as in this example), then we can add a precedence link from α to each of $\beta_1, \ldots, \beta_m$. This technique used in this example, and the resulting precedence graph is shown in Figure A.5.

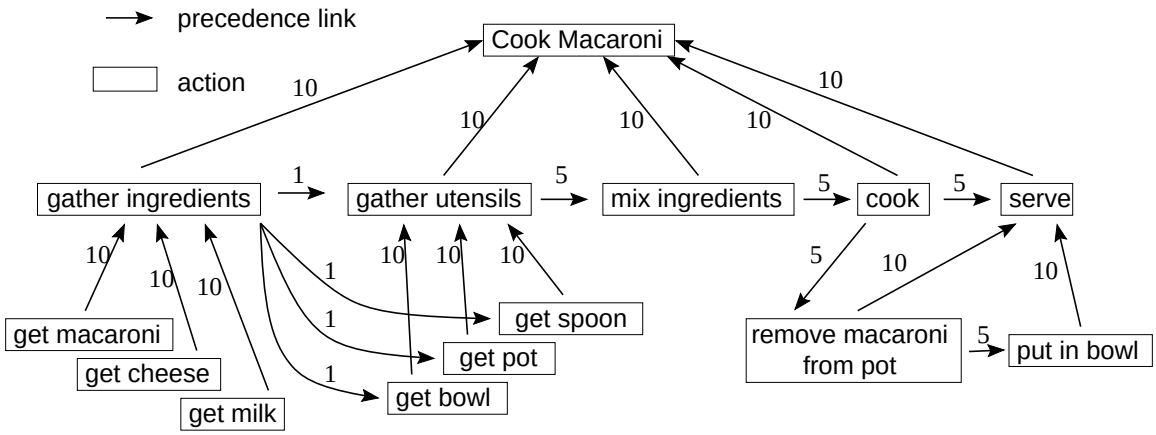


Figure A.5: Precedence of actions for Figure A.4. The original precedence links have all been given a value of 5, except for the link between "gather ingredients" and "gather utensils", which has value 1 since doing these actions out of order is not a severe problem. The decomposition links have been replaced by precedence links with a value of 10.

It is possible to derive precedence information from decomposition links by noting that for any action α, all of its subactions ought to be achieved before α. If one has a network like that in Figure A.4, then a precedence relation can be derived, as discussed in Figure A.5. Determining the penalty cost for each link is based mainly on the actions being related and how much out-of-order execution can be tolerated. In Figure A.4, there is no major problem if utensils are gathered before ingredients, so the penalty need not be large. In contrast, it makes no sense to serve the macaroni before it is cooked, so a correspondingly higher penalty should be assigned in this case. Figure A.5 shows Figure A.4 with decomposition links decomposed into their corresponding precedence links.

The *SG* system makes use of pre-assigned goal-types to determine orderings, i.e. repair-type goals precede non-repair-type goals, since the presence of a problem may invalidate an utterance. It is possible to come up with reasonable precedents among goals based on intuition and the behaviour of the program; it is usually quite clear to a

	cook macaroni	gather ingredients	gather utensils	mix ingredients	cook	serve	get macaroni	get cheese	get milk	get pot	get bowl	get spoon	remove macaroni	put in bowl
cook macaroni	·	51	40	35	30	10	61	61	61	50	50	50	25	20
gather ingredients	0	·	0	0	0	0	10	10	10	0	0	0	0	0
gather utensils	0	11	·	0	0	0	21	21	21	10	10	10	0	0
mix ingredients	0	16	5	·	0	0	26	26	26	15	15	15	0	0
cook	0	21	10	5	·	0	31	31	31	20	20	20	0	0
serve	0	41	30	25	20	·	51	51	51	40	40	40	15	10
get macaroni	0	0	0	0	0	0	·	0	0	0	0	0	0	0
get cheese	0	0	0	0	0	0	0	·	0	0	0	0	0	0
get milk	0	0	0	0	0	0	0	0	·	0	0	0	0	0
get pot	0	1	0	0	0	0	11	11	11	·	0	0	0	0
get bowl	0	1	0	0	0	0	11	11	11	0	·	0	0	0
get spoon	0	1	0	0	0	0	11	11	11	0	0	·	0	0
remove macaroni	0	26	15	10	5	0	36	36	36	25	25	25	·	0
put in bowl	0	31	20	15	10	0	41	41	41	30	30	30	5	·

Figure A.6: Final penalty matrix corresponding to the precedence graph in Figure A.4. For example, trying to mix the ingredients before getting the milk incurs 26 penalty points. The value 26 was assigned here because it is the cost of a most expensive path between the "get milk" and "mix ingredients" nodes of the precedence graph in Figure A.4.

human what goal should be performed first among the sorts of goals that SG considers. More fine-grained distinctions might be possible, but significant user-testing would be required for any particular non-obvious orderings to be preferred.

Bibliography

Allen, J. (1987). *Natural Language Understanding.* Menlo Park: The Benjamin/Cummings Publishing Company.

Allen, J., L. Schubert, G. Ferguson, P. Heeman, C. H. Hwang, T. Kato, M. Light, N. Martin, B. Miller, M. Poesio, and D. R. Traum (1995). The TRAINS project: A case study in building a conversational planning agent. *Journal of Experimental and Theoretical AI 7*, 7–48.

Ambite, J. and C. Knoblock (1997). Planning by rewriting: Efficiently generating high-quality plans. In *Proceedings of AAAI-97*, pp. 706–713.

APES (1998). Algorithms, problems, empirical studies home page. WWW: `http://www.cs.strath.ac.uk/~apes/`.

Austin, J. L. (1975). *How to Do Things with Words* (2nd ed.). Harvard University Press. Edited by J.O. Urmson and Marina Sbisà.

Bateman, J. and E. Hovy (1992). An overview of computational text generation. In C. Butler (Ed.), *Computers and Written Texts*, Chapter 3. Blackwell.

Bonet, B., G. Loerincs, and H. Geffner (1997). A robust and fast action selection mechanism for planning. In *Proceedings of AAAI-97*, pp. 714–719.

Booch, G. (1994). *Object-oriented Analysis and Design, with Applications* (2nd ed.).

Addison-Wesley.

Burke, E., K. Jackson, J. Kingston, and R. Weare (1997). Automated university timetabling: The state of the art. *The Computer Journal 40*(9), 565–571.

Burstein, M. and D. McDermott (1996). Issues in the development of human-computer mixed-initiative planning. In B. Goraskaya and J. Mey (Eds.), *Cognitive Technology: In Search of a Humane Interface*, Chapter 17, pp. 285–303. Elsevier Science.

Bylander, T. (1991). Complexity results for planning. In *Proceedings of IJCAI-91*, pp. 274–279.

Carberry, S. (1990). *Plan Recognition in Natural Language Dialogue*. MIT Press.

Carletta, J., R. Caley, and S. Isard (1993a). A collection of self-repairs from the map task corpus. Technical Report tr-47, Human Communication Research Centre, University of Edinburgh.

Carletta, J., R. Caley, and S. Isard (1993b). A system architecture for simulating time-constrained language production. Technical Report rp-43, Human Computer Research Centre, University of Edinburgh.

Cawsey, A. (1992). *Explanation and Interaction, The Computer Generation of Explanatory Dialogues*. MIT Press.

Chalmers, D. (1996). *The Conscious Mind: In Search of a Fundamental Theory*. Oxford University Press.

Chu-Carroll, J. and M. Brown (1997). Tracking initiative in collaborative dialogue interactions. In *Proceedings of ACL97*.

Cohen, P. and R. Perrault (1979). Elements of a plan-based theory of speech acts. *Cognitive Science 3*, 177–212.

Cohen, R. (1987). Analyzing the structure of argumentative discourse. *Computational Linguistics 13*(1–2), 11–24.

Cohen, R., K. Schmidt, and P. van Beek (1994). A framework for soliciting clarification from users during plan recognition. In *Proceedings of the 4th International Conference on User Modeling.*

Cormen, T., C. Leiserson, and R. Rivest (1990). *Introduction to Algorithms.* MIT Press.

Donaldson, T. (1994). A position paper on collaborative deceit. In *AAAI-94 Workshop on Planning for Interagent Communication*, pp. 33–37.

Elkan, C. (1992). Reasoning about action in first-order logic. In *Proceedings of the Conference of the Canadian Society for Computational Intelligence.*

Garey, M. and D. Johnson (1979). *Computers and Intractability, A Guide to the theory of NP-Completeness.* W.H. Freeman and Company.

Georgeff, M. (1984). A theory of action for multiagent planning. In *Proceedings of AAAI-84*, pp. 121–125.

Giacomo, G. D., Y. Lésperance, and H. Levesque (1997). Reasoning about concurrent execution, prioritized interrupts, and exogenous actions in the situation calculus. In *Proceedings of IJCAI-97*, pp. 1221–1226.

Goldman-Eisler, F. (1968). *Psycholinguistics, Experiments in Spontaneous Speech.* Academic Press.

Grosz, B. and C. Sidner (1985). Discourse structure and the proper treatment of interruptions. In *Proceedings of IJCAI-85*, pp. 832–839.

Grosz, B. and C. Sidner (1986). Attentions, intentions, and the structure of discourse. *Computational Linguistics 12*(3), 175–204.

Grosz, B. and C. Sidner (1990). Plans for discourse. In P. R. Cohen, J. Morgan, and M. E. Pollack (Eds.), *Intentions in Communications*, Chapter 20, pp. 416–444. MIT Press.

Gu, J. (1992). Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin 3*(1), 8–12.

Guinn, C. (1993). A computational model of dialogue initiative in collaborative discourse. In *Human-Computer Collaboration: Reconciling Theory, Synthesizing Practice, Papers from the 1993 Fall Symposium Series.* AAAI Technical Report FS-93-05.

Haller, S. and S. McRoy (Eds.) (1997a). *Computational Models for Mixed Initiative Interaction.* Working notes.

Haller, S. and S. McRoy (Eds.) (1997b). *Working Notes of the AAAI-97 Spring Symposium on Computational Models for Mixed Initiative Interaction.*

Halliday, M. (1985). *Spoken and Written Language.* Deakin University Press.

Hirschberg, J. and D. Litman (1993). Empirical studies on the disambiguation of cue phrases. *Computational Linguistics 19*(3), 501–530.

Hirst, G. (1987). *Semantic Interpretation and the Resolution of Ambiguity.* Cambridge University Press.

Hovy, E. (1993). Automated discourse generation using discourse structure relations. *Artificial Intelligence 63*(1–2), 341–385.

Hovy, E. and K. McCoy (1989). Focusing your RST: A step toward generating coherent multisentential text. In *Eleventh Annual Conference of the Cognitive Science Society*, pp. 667–674. Lawrence Erlbaum.

Joshi, A., B. Webber, and R. Weischedel (1984). Living up to expectations: Computing

expert responses. In *Proceedings of AAAI-84*, pp. 169–175.

Kautz, H. and B. Selman (1996). Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of AAAI-96*.

Kitano, H., M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara (1997). RoboCup, a challenge problem for ai. *AI Magazine 18*(1), 73–85.

Knight, K. and I. Chander (1994). Automated postediting of documents. In *Proceedings of AAAI-94*, pp. 779–784.

Kolodner, J. (1993). *Case Based Reasoning*. Morgan Kaufmann.

Kukich, K. (1992). Techniques for automatically correcting words in text. *Computing Surveys 24*(4), 377–439.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AI Magazine 13*(1), 32–44.

Lambert, L. and S. Carberry (1991). A tripartite plan-based model of dialogue. In *Proceedings of the Twenty-ninth Annual Meeting of the Association for Computational Linguistics*, pp. 47–54.

Landauer, T. (1995). *The Trouble with Computers, Usefulness, Usability, and Productivity*. MIT Press.

Levesque, H., R. Reiter, Y. Lespérance, F. Lin, and R. Scherl (1997). GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming 31*, 59–84.

Lin, S. and B. Kernighan (1973). An effective heuristic algorithm for the travelling salesman problem. *Operations Research 21*, 498–516.

Litman, D. and J. Allen (1990). Discourse processing and commonsense plans. In P. R.

Cohen, J. Morgan, and M. E. Pollack (Eds.), *Intentions in Communication*, Chapter 17, pp. 365–388. The MIT Press.

Maes, P. (1990). Situated agents can have goals. *Journal for Robotics and Autonomous Systems 6*(1), 49–70.

Mann, W. and S. Thompson (1988). Rhetorical structure theory: Toward a functional theory of text organization. *Text 3*, 243–281.

Mauldin, M. (1994). ChatterBots, TinyMuds, and the Turing Test: Entering the Loebner Prize competition. In *Proceedings of AAAI-94*, pp. 16–21.

McCarthy, J. and P. Hayes (1969). Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie (Eds.), *Machine Intelligence 4*, pp. 463–502. Edinburgh University Press.

McCoy, K. and J. Cheng (1991). Focus of attention: Constraining what can be said next. In C. Paris, W. Swartout, and W. Mann (Eds.), *Natural Language Generation in Artificial Intelligence and Computational Linguistics*, Chapter 4. Kluwer Academic.

McKeown, K. (1985). *Text Generation, Using discourse strategies and focus constraints to generate natural language text.* Cambridge University Press.

Minton, S., M. Johnston, A. Philips, and P. Laird (1992). Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence 58*, 161–205.

Moore, J. (1995). *Participating in explanatory dialogues : interpreting and responding to questions in context.* MIT Press.

Moore, J. and M. Pollack (1992). A problem for RST. *Computational Linguistics 18*(4), 537–544.

Nielsen, J. (1993). *Usability Engineering.* Academic Press.

Norvig, P. (1992). *Paradigms of Artificial Intelligence, Case Studies in Common Lisp.* Morgan Kaufmann.

Oreström, B. (1983). *Turn-taking In English Conversation.* CWK Gleerup.

O'Shaughnessy, D. (1995). Approaches to improve automatic speech synthesis. In R. Ramachandran and R. Mammone (Eds.), *Modern Methods of Speech Processing*, Chapter 14, pp. 325–347. Kluwer.

Paris, C. (1993). *User Modelling In Text Generation.* Pinter.

Passonneau, R. and D. Litman (1993). Intention-based segmentation: Human reliability and correlation with linguistic clues. In *Proceedings of the Conference of the 31st Annual Meeting of the ACL*, pp. 148–155. Ohio.

Pearl, J. (1984). *Heuristics, Intelligent Search Strategie for Intelligent Problem Solving.* Addison-Wesley.

Pollack, M., J. Hirschberg, and B. Webber (1982). User participation in the reasoning process of expert systems. In *Proceedings of AAAI-82*, pp. 358–361.

Ramachandran, R. and R. Mammone (Eds.) (1995). *Modern Methods of Speech Processing.* Kluwer.

Reddy, R. (1990). Speech recognition by machine: A review. In A. Waibel and K. Fu Lee (Eds.), *Readings in Speech Recognition*, pp. 8–38. Morgan Kaufmann.

Reinelt, G. (1994). *The Traveling Salesman, Computational Solutions for TSP Applications.* Springer-Verlag.

Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz (Ed.),

Artificial Intelligence and Mathematical Theory of Computation, Papers in Honor of John McCarthy, pp. 359–280. Academic Press.

Rich, C. and C. Sidner (1998). COLLAGEN: A toolkit for building collaborative interface agents. *User Modeling and User-adapted Interaction*. To appear.

Russell, S. and P. Norvig (1995). *Artificial Intelligence, A Modern Approach*. Prentice Hall.

Sacks, H., E. Schegloff, and G. Jefferson (1978). A simplest systematics for the organization of turn taking for conversation. In J. Schenkein (Ed.), *Studies in the Organization of Conversational Interaction*, Chapter 1, pp. 7–55. Academic Press.

Searle, J. (1970). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press.

Selman, B., H. Kautz, and B. Cohen (1994). Noise strategies for improving local search. In *Proceedings of AAAI-94*, pp. 337–343.

Selman, B., H. Levesque, and D. Mitchell (1992). A new method for solving hard satisfiability problems. In *Proceedings of AAAI-92*, pp. 440–446.

Shieber, S. (1994). Lessons from a restricted turing test. *Communications of the ACM 37*(6), 70–78. Also available from the Center for Research in Computing Technology, Harvard University, as Technical Report TR-19-92.

Shoham, Y. (1993). Agent-oriented programming. *Artificial Intelligence 60*(1), 51–92.

Sidner, C. (1994). An artificial discourse language for negotiation. In *Proceedings of AAAI 94*, pp. 814–819.

Smith, R., D. Hipp, and A. Biermann (1995). An architecture for voice dialog systems based on prolog-style theorem proving. *Computational Linguistics 21*(3), 281–320.

Traum, D. and E. Hinkelman (1992). Conversation acts in task-oriented spoken dialogue. *Computational Intelligence 8*(3), 575–599.

Tsang, E. (1993). *Foundations of Constraint Satisfaction*. Academic Press.

van Beek, P., R. Cohen, and K. Schmidt (1993). From plan critiquing to clarification dialogue for cooperative response generation. *Computational Intelligence 9*(2), 132–154.

Waibel, A. and K. Fu Lee (Eds.) (1990). *Readings in Speech Recognition*. Morgan Kaufmann.

Walker, M. (1993). *Informational Redundancy and Resource Bounds in Dialogue*. Ph. D. thesis, University of Pennsylvania.

Walker, M. (1994). Experimentally evaluating communicative strategies: The effect of the task. In *Proceedings of AAAI-94*, pp. 86–93.

Walker, M., D. Hindle, J. Fromer, G. D. Fabbrizio, and C. Mestel (1997). Evaluating competing agent strategies for a voice email agent. In *Proceedings of the 5th European Conference on Speech Technology and Communication, EUROSPEECH 97*.

Walker, M. and S. Whittaker (1990). Mixed initiative in dialogue: An investigation into discourse segmentation. In *Proceedings of the 28th ACL Conference*, pp. 70–78. Pittsburgh.

Wallace, R. and E. Freuder (1995). Anytime algorithms for constraint satisfaction and sat problems. In *IJCAI-95 Workshop on Anytime Algorithms and Deliberation Scheduling*.

Weizenbaum, J. (1966). Eliza. *Communications of the ACM 9*, 36–45.

Weld, D. (1994). An introduction to least commitment planning. *AI Magazine 15*(4),

27–61.

Whittaker, S. and P. Stenton (1988). Cues and control in expert-client dialogues. In *Proceedings of the 26th ACL Conference*, pp. 123–130. Buffalo.

Winograd, T. (1983). *Language as a Cognitive Process*. Massachusetts: Addison-Wesley.

Wooldridge, M. and N. Jennings (1995). Intelligent agents: Theory and practice. *Knowledge Engineering Review 10*(2).

Young, S., A. Hauptmann, W. Ward, E. Smith, and P. Werener (1990). High level knowledge sources in usable speech recognition systems. In A. Waibel and K. Fu Lee (Eds.), *Readings in Speech Recognition*, pp. 538–549. Morgan Kaufmann.