Towards Tractable Agent-based Dialogue

by

Nathan James Blaylock

Submitted in Partial Fulfillment
of the
Requirements for the Degree
Doctor of Philosophy

Supervised by

Professor James F. Allen

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2005

Curriculum Vitae

Nate Blaylock was born in Provo, Utah on June 8, 1975, and grew up in the neighboring city of Orem. He attended Brigham Young University (BYU) from 1993 to 1994 and then took two years off to serve as a missionary for the Church of Jesus Christ of Latter-day Saints in the Japan Tokyo North Mission. While a missionary, Nate learned Japanese, which, as an agglutinating language, was so regular, he often thought that verb conjugations could be done by a computer.

Upon completing his mission in 1996, he reentered BYU and took a Linguistics class "on a whim." It was in this class that he was first introduced to Computational Linguistics, which prompted him to add a Linguistics major to his Computer Science major. He graduated from BYU in 1999 with a BS in Computer Science and a BA in Linguistics.

Nate entered the PhD program at the University of Rochester's Department of Computer Science in the Fall of 1999 and actively worked in the TRIPS research group (with advisor James Allen) during his time in Rochester. He was granted an MS in Computer Science in 2001.

In the Spring of 2004, Nate accepted a position as a research associate in the Department of Computational Linguistics, Saarland University, in Saarbrücken, Germany, where he currently working on the EU-funded TALK project.

Acknowledgments

I would first like to thank my advisor, James Allen. He gave me the freedom to work on what interested me and at the same time guidance to steer my research in the right direction. I would also like to thank my thesis committee members, Greg Carlson, Henry Kyburg, and Len Schubert for their comments and suggestions.

I have done this work in two very rich research environments. Working with the people in the TRIPS group at Rochester — including James Allen, George Ferguson, Amanda Stent, Donna Byron, Lucian Galescu, Myrosia Dzikovska, Joel Tetreault, Scott Stoness, Ellen Campana, Greg Aist, Mary Swift, Carlos Gómez-Gallo, Phil Michalak, and Nate Chambers — was a great experience. They were a great group of people to work with, both personally and professionally.

For the last year, I have been also working with the Saarbrücken TALK group — Manfred Pinkal, Ivana Kruijff-Korbayová, Ciprian Gerstenberger, Verena Rieser, Tilman Becker, Peter Poller, Jan Schehl, Michael Kaißer, Gerd Fliedner, Daniel Bobbert and Diana Steffen — which has been equally rewarding. The group has always been willing to listen to and comment on my ideas.

I also had the opportunity to do several summer internships, where I was able to work with great people and expand my horizons. I'd like to especially thank Manfred Pinkal, Steve Richardson, and John Dowding for making those internships possible, as well as all the people I was able to work with.

I am deeply indebted to the Rochester staff, who were always there to pull me out of last-minute problems, and were still nice enough to chat with me when I didn't have

crises. I would especially like to thank Elaine Heberle, Marty Guenther, Peg Meeker, Jill Forster, Eileen Pullara, and JoMarie Carpenter.

My parents, Nan and Giff Blaylock have always supported my educational goals and were always there to encourage me when I needed it. I would also like to thank my brother Seth Blaylock who proof-read this thesis for me.

Finally, and most importantly, I would like to thank my wife Felicita, without whom I may have never finished. She was always there to tell me to take it easy when I needed it, to nudge me when I needed extra motivation, and to make me laugh, even when I didn't feel like it. *Gracias, mi perla.*

This thesis is based upon work supported by a grant from DARPA (no. #F30602-98-2-0133); a grant from the Department of Energy (no. #P2100A000306); two grants from The National Science Foundation (award #IIS-0328811 and award #E1A-0080124); and the EU-funded TALK Project (no. IST-507802). Any opinions, findings, conclusions or recommendations expressed in this thesis are those of the author and do not necessarily reflect the views of the above-named organizations.

Abstract

This thesis describes research which attempts to remove some of the barriers to creating true conversational agents — autonomous agents which can communicate with humans in natural language. First, in order to help bridge the gap between research in the natural language and agents communities, we define a model of agent-agent collaborative problem solving which formalizes agent communication at the granularity of human communication. We then augment the model to define an agent-based model of dialogue, which is able to describe a much wider range of dialogue phenomena than plan-based models. The model also defines a declarative representation of communicative intentions for individual utterances.

Recognition of these intentions from utterances will require an augmentation of already intractable plan and intention recognition algorithms. The second half of the thesis describes research in applying statistical corpus-based methods to goal recognition, a special case of plan recognition.

Because of the paucity of data in the plan recognition community, we have generated two corpora in distinct domains. We also define an algorithm which can stochastically generate artificial corpora to be used in learning. We then describe and evaluate fast statistical algorithms for both flat and hierarchical recognition of goal schemas and their parameter values. The recognition algorithms are more scalable than previous work and are able to recognize goal parameter values as well as schemas.

Table of Contents

Curriculum Vitae ii

Acknowledgments iii

Abstract v

List of Tables ix

List of Figures xi

1 Introduction 1

1.1 Agent-based Dialogue Modeling 3

1.2 Requirements for Agent-based Dialogue 4

1.3 Thesis Overview 8

2 Dialogue Modeling: Background 13

2.1 Fixed-Task Models 13

2.2 Plan-based Models 16

3 A Model of Collaborative Problem Solving 31

3.1 The Collaborative Problem-Solving Process 32

3.2 A Collaborative Problem-Solving Model 36

3.3 Single-agent Problem Solving 37

3.4 Collaborative Problem Solving 63

3.5 Conclusions and Future Work 73

4 Modeling Dialogue as Collaborative Problem Solving 75

4.1 Collaborative Problem Solving and Communicative Intentions 76

4.2 Grounding 78

4.3 Coverage of the CPS Dialogue Model 89

4.4 Conclusions and Future Work 98

5 Plan Recognition: Background 100

5.1 Intention Recognition and Plan Recognition 101

5.2 Requirements for Plan Recognition 102

5.3 Previous Work in Plan Recognition 103

5.4 Goal Recognition 108

5.5 Towards Statistical Goal Recognition 114

6 Obtaining Corpora for Statistical Goal Recognition 116

6.1 Definitions 117

6.2 Existing Plan Corpora 117

6.3 The Linux Corpus 119

6.4 General Challenges for Plan Corpora Collection 125

6.5 Generating Artificial Corpora 128

6.6 The Monroe Corpus 133

6.7 Plan Corpora: Human vs. Artificial 136

6.8 Conclusions and Future Work 138

7 Flat Goal Recognition **139**

7.1 Problem Formulation 139

7.2 Goal Schema Recognition 142

7.3 Goal Parameter Recognition 152

7.4 Instantiated Goal Recognition 163

7.5 Conclusion 167

8 Hierarchical Goal Recognition **169**

8.1 Goal Schema Recognition 170

8.2 Goal Parameter Recognition 189

8.3 Instantiated Goal Recognition 196

8.4 Conclusion 203

9 Conclusion **205**

9.1 Future Work in Dialogue Modeling 207

9.2 Future Work in Goal Recognition 209

9.3 Future Work in Agent-based Dialogue Systems 212

Bibliography **215**

A Instructions Given to Users in the Linux Corpus Collection **228**

B Goal Schemas in the Linux Corpus **230**

C Action Schemas in the Linux Corpus **232**

C.1 General Issues for Conversion 232

C.2 The Action Schemas 235

D Goal Schemas in the Monroe Corpus **237**

List of Tables

3.1 Abbreviations for Type and Feature Names 68

4.1 Conversation Act Types [Traum and Hinkelman1992] 80

6.1 Contents of the Linux Corpus 124

6.2 Comparison of the Linux and Monroe Corpora 134

7.1 Goal Schema Recognition Results on the Monroe Corpus 150

7.2 Goal Schema Recognition Results on the Linux Corpus 151

7.3 Goal Parameter Recognition Results on the Monroe Corpus 161

7.4 Goal Parameter Recognition Results on the Linux Corpus 162

7.5 Instantiated Goal Recognition Results for the Monroe Corpus 166

7.6 Instantiated Goal Recognition Results for the Linux Corpus 167

8.1 Results of Schema Recognition using the CHMM 185

8.2 Results of Flat Schema Recognition on the Monroe Corpus (from Chapter 7) 185

8.3 Results of Schema Recognition using the CHMM and Observation Information 188

8.4 Results of Parameter Recognition 194

8.5 Results of Flat Parameter Recognition on the Monroe Corpus (from Chapter 7) 195

8.6 Results of Instantiated Recognition 201

8.7 Results of Flat Instantiated Recognition on the Monroe Corpus (from Chapter 7) 202

B.1 Goal Schemas in the Linux Corpus 231

D.1 Goal Schemas in the Monroe Corpus 237

List of Figures

1.1 Example Dialogue 2

1.2 Possible Architecture of an Agent-based Dialogue System 4

3.1 Type Description for *object* 39

3.2 Type Description for *ps-object* 41

3.3 Type Description for *slot* 43

3.4 Type Description for *single-slot* 43

3.5 Type Description for *filler* 44

3.6 Type Description for *filler(*σ*)* 44

3.7 Type Description for *single-slot(*σ*)* 45

3.8 Type Description for *multiple-slot* 45

3.9 Type Description for *constraints-slot* 45

3.10 Type Description for *evaluations-slot* 46

3.11 Type Description for *objectives-slot* 47

3.12 Type Description for *objective* 47

3.13 Full Type Description for *objective* 48

3.14 Type Description for *recipe* 48

3.15 Type Description for *constraint* 49

3.16 Type Description for *resource* 49

3.17 Type Description for *evaluation* 50

3.18 Type Description for *situation* 50

3.19 Type Description for *song* 53

3.20 Type Description for *listen-song* 53

3.21 A Simple Initial PS State 57

3.22 The PS State after Executing *identify-objective(actual-situation,*[2]*)* . . . 60

3.23 Type Description for *c-situation* 66

3.24 *listen-song* Objective [2] 69

3.25 The CPS State after A's First Turn 70

3.26 The Abbreviated Version of Figure 3.25 71

3.27 The CPS State after B's First Turn 72

3.28 *song* Resource [4] — "Yesterday" by the Beatles 72

3.29 The CPS State after A's Second Turn 73

4.1 Example of Conversation Acts: Grounding Acts [Traum and Hinkelman1992] 82

4.2 Example of Conversation Acts: Core Speech Acts [Traum and Hinkelman1992] 82

4.3 The TRAINS Example Interpreted with the Agent-based Model 85

4.4 Contents of *objective* [1] 87

4.5 Contents of *recipe* [2] 88

4.6 Contents of *constraint* [3] 88

4.7 A Planning Dialogue from [Traum and Hinkelman1992] (Continuation of Figure 4.3) 91

4.8 Execution Dialogue from [Grosz and Sidner1986]: Part 1 93
4.9 Execution Dialogue from [Grosz and Sidner1986]: Part 2 94
4.10 A Planning and Execution Dialogue (from Figure 1.1) 97

7.1 Schema Recognition Example: $\tau = 0.8$ 145
7.2 Evaluation Metrics for Example in Figure 7.1 149

8.1 An Example Plan Tree 171
8.2 A Cascading Hidden Markov Model (CHMM) 173
8.3 Algorithm for Calculating Forward Algorithm for CHMMs 176
8.4 The Sequence of Goal Chains Corresponding to the Plan Tree in Figure 8.1 178
8.5 A Plan Tree with Varying Depth 179
8.6 The Expanded Version of the Plan Tree in Figure 8.5 180
8.7 The Sequence of Goal Chains Corresponding to the Expanded Plan Tree in Figure 8.6 180

1 Introduction

Language is typically used as a means to an end. People use language to help them achieve their goals by soliciting help from and coordinating their efforts with those with whom they talk.

The range of activities that can be accomplished (or supported) through language is vast. The following are just a few examples:

1. Obtaining (and providing) information

   A: where is the bathroom?
   B: up the stairs to your left.

2. Getting someone to perform some action (and report the result)

   A: turn off the sprinklers.
   B: [turns off sprinklers]
   done.

3. Getting someone to suggest a course of action

   A: how can I open this door?
   B: push the red button, wait 30 seconds, and then twist the knob.

The above are short dialogue exchanges illustrating some of the individual uses of language. However, dialogue exchanges can be much longer and freely mix the different individual uses as dialogue partners work together to achieve a goal.

1.1 A: let's go to the park today.
2.1 B: okay.
2.2 B: should we walk or drive?
3.1 A: what's the weather going to be like?
4.1 B: I don't know.
4.2 B: let's watch the weather report.
5.1 A: no, it's not on until noon.
5.2 A: just look on the internet.
6.1 B: okay. [looks on internet]
6.2 B: it's supposed to be sunny.
7.1 A: then let's walk.
8.1 B: okay.
9.1 A: do you want to go now?
10.1 B: sure.

Figure 1.1: Example Dialogue

Consider the dialogue in Figure 1.1. Here the dialogue participants use language to agree on a common goal (going to the park); to decide on a plan for accomplishing the goal (walking); to coordinate execution of the plan (when to leave the house); and finally, to plan and execute a separate goal in order to help them decide on a plan (looking up weather on the internet).

Although this type of dialogue is fairly common, we are unaware of any dialogue system that could handle it in a general way. Current models of dialogue do not handle the level of complexity shown in Figure 1.1. Most dialogue systems use dialogue models tailored to a specific application. Other, more general systems model dialogue as coordination in joint execution of predetermined plans (e.g., [Cohen et al.1991; Ardissono, Boella, and Lesmo1996; Rich, Sidner, and Lesh2001]) or coordination in building joint plans, but not their execution (e.g., [Grosz and Sidner1990; Ferguson and Allen1998; Chu-Carroll and Carberry2000]). Both of these general approaches are typically referred to as "plan-based" dialogue models, as they use generalized mechanisms which work on a planning representation of actions and goals in the domain.

Although they are domain-independent, these plan-based dialogue models still can-

not cover the type of dialogue in Figure 1.1, primarily because it consists of both planning *and* execution. Also, most models would also not be able to handle the beginning interchange of the sample dialogue, when the participants agree on a goal. Most plan-based models assume that a goal is already mutually known and agreed-upon *before* the dialogue begins.

Our goal is to build dialogue systems capable of engaging in the what Allen et al. call *practical dialogue* — "dialogue focused on accomplishing some specific task," [Allen et al.2000]. We, as they, believe that this "genre" of dialogue is what humans will most want to use with machines (as opposed to, say, humorous or social dialogue).

1.1 Agent-based Dialogue Modeling

In this thesis, we present an *agent-based* approach to dialogue systems, in which we model dialogue as *collaborative problem solving* between agents. Before we go any further, we will define what we mean by some of these terms.

First, *problem solving* is the general process used by agents of formulating and pursuing goals. It can include activities such as goal selection (or abandonment); the selection of plans as courses of action for accomplishing a goal; the execution and monitoring of plans; replanning (when errors occur); and the cognitive processes (such as goal evaluation) used for making these decisions. Problem solving is closely related to the concept of "rational behavior" (cf. [Cohen and Levesque1990a; Rao and Georgeff1995]), and can be described as the driving core of an agent. *Collaborative problem solving* is the coordinated problem solving of two or more agents.

There are several things which make collaborative problem solving an attractive model for dialogue. First, it subsumes previous plan-based models of dialogue, since it includes both planning and execution. However, as described above, collaborative problem solving goes beyond just planning and execution; it attempts to describe the full set of agent activities. We believe that humans are also (to some approximation)

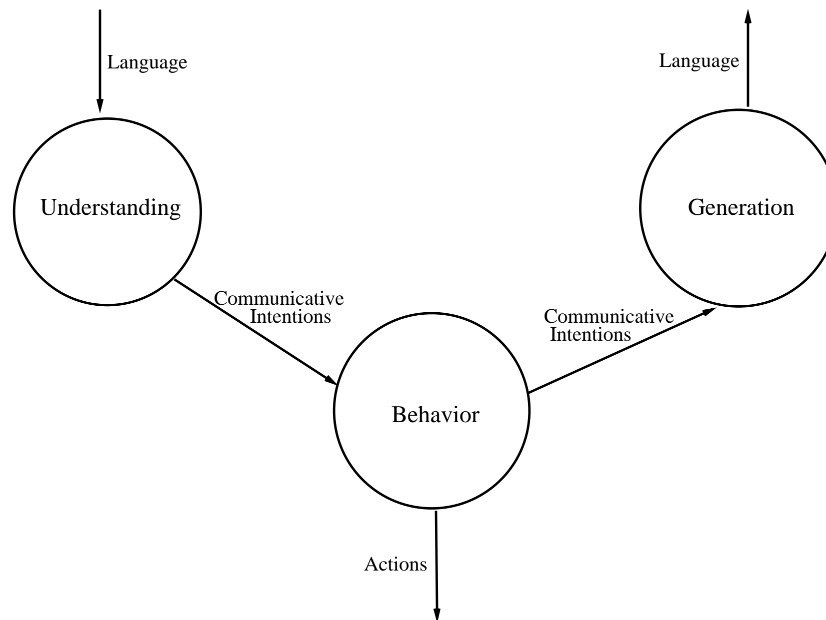


Figure 1.2: Possible Architecture of an Agent-based Dialogue System

collaborative problem-solving agents, and that most practical dialogue can be modeled by collaborative problem solving.

1.2 Requirements for Agent-based Dialogue

A dialogue model is just the first step to building a dialogue system. Making generalized, agent-based dialogue systems a reality will require progress in many areas of artificial intelligence, including natural language processing and autonomous agents. In this section we first describe a generalized architecture for an agent-based dialogue system based on the TRIPS dialogue system architecture [Allen, Ferguson, and Stent2001; Blaylock, Allen, and Ferguson2002]. Then, using that architecture as a reference point, we describe some requirements of an agent-based dialogue system.

The architecture shown in Figure 1.2 divides the dialogue system into three subsystems: *interpretation*, *behavior*, and *generation*. In interpretation, the system must be

able to *understand* what is meant by a user's utterance. This goes much deeper than just what the utterance means on the surface. The system must be able to understand much more. Why did the user utter what he did? What was he trying to accomplish by uttering it? These conversational "ends" are called *communicative intentions*.

The behavioral subsystem is essentially an intelligent agent that is able to reason with the user's communicative intentions, (as well as other things such as the world state, its own goals and intentions, etc.) and decide what actions to take (either in the world or cognitively). It also needs to be able to form its own intentions for communication with the user.

In the generation subsystem, the system's communicative intentions are transformed into natural language and uttered to the user.

As mentioned above, there are many aspects to our ideal agent-based system that the current state-of-the-art does not support. We discuss several of the most prominent by subsystem.

1.2.1 Interpretation

The task of interpretation is typically divided into many levels of analysis (speech recognition/synthesis, morphology, syntax, semantics, pragmatics, etc.), and an enormous body of research has been done at all of them. The most salient level, however, for agent-based dialogue is at the "topmost" level, which we call the *intention/language interface*, where communicative intentions are recognized from a high-level semantic form (i.e., communicative acts). This process of inferring communicative intentions from communicative acts is called *intention recognition*.

There are several barriers to using current intention recognition techniques in an agent-based system. First, we need a representation of the communicative intentions to be recognized. Second, we need algorithms that can recognize intentions for the

full range of collaborative problem solving. Lastly, such algorithms need to be fast, as dialogue happens in real time. We discuss each of these below.

Representation of Communicative Intentions Very little work has been done on descriptive representations of communicative intentions. Most dialogue systems do not use intention recognition — the domains covered are usually simple and the exchanges circumscribed, so that intentions are transparent from the utterance semantics. In systems which do perform intention recognition (typically plan-based systems), communicative intentions are not explicitly represented, but rather are tied to the intention algorithm which updates some sort of dialogue state.

In Chapter 4, we present a descriptive language of communicative intentions based on the model of collaborative problem solving.

Intention Recognizer for Collaborative Problem Solving As mentioned above, in most dialogue systems, explicit intention recognition is not needed, as the domain and exchanges are simple and transparent. Plan-based dialogue systems, however — especially those that model dialogue as joint planning — often do perform intention recognition. However, these intention recognition algorithms are specifically tied to the dialogue model (typically planning), and would not be directly applicable to an agent-based dialogue model. It is possible (and indeed we believe it to be the case), that these algorithms can be extended to cover the range of collaborative problem solving, but such a question is left to future research.

Real-time Intention Recognition Another problem with intention recognition is speed. Dialogue happens in real time, and the system is expected to respond to the user within a reasonable amount of time (on the order of several seconds). However, all intention recognizers of which we are aware are based on the more general process of *plan recognition* — the recognition of an agent's plan from observed actions. In the gen-

eral case, plan recognition is known to be exponential in the number of possible goals [Kautz1991], thus for any reasonable-sized domain, intention recognition will be too slow to support real-time dialogue. Although more recent work has improved on this (e.g., [Vilain1990]), it has done so at the expense of expressiveness and flexibility.

Unless alternative methods of intention recognition can be found, faster, more scalable methods of plan recognition will be essential to supporting real-time dialogue for agent-based systems.

In the second half of this thesis, we present a goal recognizer — a special type of plan recognizer — which preserves scalability without a large sacrifice of expressiveness, which could be used as the main engine for a real-time intention recognizer for agent-based dialogue.

1.2.2 Behavior

Once an utterance has been successfully interpreted, the behavioral subsystem must be able to act on it. The behavioral system must be an intelligent agent that is not only able to reason and act in the world (i.e., do problem solving), it must also be able to communicate and coordinate its activities with a (human) user (i.e., do collaborative problem solving).

Much work has been done on building autonomous agents, but very little on human-agent collaboration. In fact, most agent-agent collaboration work makes explicit assumptions that both agents are artificial (assuming, for example, than the two collaborating agents are running the same code). This assumption, however, cannot be made when one of those agents is a human!

In Chapter 3, we present a model of collaborative problem solving which is based on human communication, and show how this can be used to model human-agent or agent-agent communication. This is a first step to building agents that can collaborate with humans in a human-like way.

Work still remains, however, to actually build an autonomous agent that uses and reasons with the collaborative problem-solving model.

1.2.3 Generation

Parallel to interpretation, work on generation has been done at many levels (phonology, syntax, etc.) The most important for agent-based dialogue, of course, is that of converting communicative intentions to some high-level semantic form (a process we will term *content planning*).

This is perhaps the most open of the areas mentioned here. Most generation systems for dialogue assume that the behavioral component (usually called the *dialogue manager*) has already transformed intentions to a semantic form, and we are unaware of any system that generates directly from communicative intentions (probably because most systems do not have an explicit representation for these intentions).

1.3 Thesis Overview

As noted in the preceding section, building an agent-based dialogue system requires a lot of work in many areas. Unfortunately, the required amount of research is outside the scope of a single thesis, and we do not attempt it here. Instead, as the title suggests, this thesis represents our efforts to provide several of the necessary foundational pieces required for further work on agent-based dialogue systems.

The remainder of this thesis can be divided into two parts, representing its two main areas of contributions. The first part (Chapters 2 to 4) describes a domain-independent model of dialogue based on agent collaborative problem solving. The second part (Chapters 5 to 8) presents a fast algorithm for goal recognition — a special type of plan recognition. Chapter 9 then concludes the thesis and discusses areas of future work.

We now discuss each of these main contributions in more detail.

1.3.1 An Agent-based Dialogue Model

As stated above, the most vital foundation to supporting agent-based dialogue is a dialogue model. In Chapter 2, we introduce the field of dialogue modeling and discuss previous work.

In Chapter 3, we present a model of collaborative problem solving for agents. In this model an agent (individually) or group of agents (collaboratively) perform problem-solving activities in three general areas.

- *Determining Objectives*: In this area, agents manage their objectives, deciding to which they are committed, which will drive their current behavior, etc.
- *Determining and Instantiating Recipes for Objectives*: In this area, agents choose recipes to use towards attaining their objectives. Agents may either choose and instantiate recipes from a recipe library, or they may choose to *create* new recipes via planning.
- *Acting*: In this area, agents follow a recipe and execute atomic actions. This area also includes monitoring the execution to check for success.

It is important to note that these areas need not proceed in order and may be freely interleaved. Decisions made at one level can also be revisited at any point. Collaborative decisions are negotiated among the agents, and only become valid when both agents agree to a proposal.

This model is able to describe a large range of collaborative agent activity, including not only (interleaved) planning and execution, but also goal selection, replanning, evaluations, etc. This general collaborative problem-solving model contributes not only

to the study of human-agent dialogue, but is also general enough to be applied to general collaboration among heterogeneous agents, whether human or artificial, through natural language or by other means.

In Chapter 4, we use this collaborative problem-solving model as the basis for an agent-based model of dialogue. In particular, we model dialogue participants as collaborative agents, and the dialogue itself as the collaboration about their problem-solving activities. Because this dialogue model is based on a rich model of agent collaboration, it is able to describe a much wider range of dialogue types than previous models, including dialogues similar to that shown in Figure 1.1.

An additional contribution described in Chapter 4 is a domain-independent, descriptive language for communicative intentions. The communicative intentions of an utterance are modeled as attempts to negotiate collaborative problem-solving decisions. As discussed above, most dialogue systems treat communicative intentions only implicitly in language understanding and intention recognition algorithms. This description language supports the labeling of an utterance with its intended meaning and lays the foundation for future work in agent-based intention recognition algorithms (see Chapter 9).

1.3.2 Fast Goal Recognition

The other major area of contribution of this thesis is that of using machine learning techniques to build fast goal recognition algorithms. As discussed in Section 1.2.1, fast goal recognition is important part of building a real-time intention recognizer to support agent-based dialogue.

Plan recognition is the general process of inferring an agent's current plan based on observed actions. Goal recognition is a special case of plan recognition where only the agent's current goals are inferred. In Chapter 5, we introduce plan and goal recognition and previous work in this area.

In order to support machine learning, one must have corpora, and there is a paucity of such for plan recognition. In Chapter 6, we describe methods we used in collecting two labeled *plan corpora*.

The first corpus, the Linux corpus, was gathered by giving human Linux users a goal and then recording their commands used to achieve it, similar to the collection done for the Unix corpus [Lesh1998]. This data collection resulted in the contribution of a corpus of more than 400 goal-labeled action sequences — more than an order of magnitude larger than the Unix corpus.

However, in many domains, labeled corpora are difficult to gather, especially corpora that include hierarchical goal structure (see below). For the second corpus, we introduce a novel technique for automatically generating labeled plan corpora. We use an AI planner to stochastically generate plan sequences for generated goals and start states. This provides a way to rapidly generate data for machine learning in plan recognition in domains where human data is difficult or impossible to obtain. The resulting Monroe corpus contains 5000 hierarchical plans for an disaster-management domain. Together with the Linux corpus, this contributes two labeled corpora as a resource for researchers in plan recognition (something that was lacking before, as observed in [Lesh1998]). In addition, the stochastic plan corpus generation techniques can be used to quickly generate corpora in other domains as well.

In Chapter 7, we describe a *flat* goal recognizer (a recognizer which only recognizes a current top-level goal, but no subgoals). The recognizer is able to perform fast recognition of both goal schemas and their parameter values. It achieves reasonable results on both the Linux and Monroe corpora and its time complexity is only linear in the number of goal schemas and observed actions, making it much faster than previous systems.

In Chapter 8, we augment the flat recognizer to perform hierarchical goal recognition — recognizing both the top-level goal as well as active subgoals. The recognizer uses cascading Hidden Markov Models to compute the most likely subgoal at each

level of the hierarchy. It is fast, with a time complexity quadratic in the number of goal schemas and linear in the number of subgoal levels, and gives encouraging results on test cases in the Monroe domain. This contributes a fast, flexible hierarchical goal recognizer which can be rapidly ported to new domains. We expect to use this recognizer as the principal component of an intention recognizer in future work (see Chapter 9).

2 Dialogue Modeling: Background

The goal of *dialogue modeling* is to provide a representation of dialogue processes which can be used to build systems that act as dialogue participants. Such a representation is an important first step towards building a dialogue system, especially for performing *dialogue management* — the process of controlling the overall dialogue behavior of the system.

In this chapter we discuss previous work in dialogue modeling. We first discuss fixed-task dialogue models and then plan-based models.

2.1 Fixed-Task Models

In this section, we group together three classes of dialogue models which make the (explicit or implicit) assumption that the form of the *domain task* (the non-linguistic task which is the focus of the dialogue) is known (and encoded) at the design time of the dialogue system. As we discuss below, this does *not* necessarily mean that the form of the dialogue itself (the *discourse task*) need be fixed.

For fixed-task systems, a prototypical dialogue task is that of database lookup for such information as flight or train schedules [Rudnicky et al.1999; Lamel et al.2000] or weather reports [Zue et al.2000]. Here, the form of the domain task (the database

lookup) is fixed, and the system only needs to query the user for certain information, such as the user's preferences or personal details, in order to be able to perform the task.

We will see in the section below that this constraint is removed by plan-based models, which allows (to varying degrees) the user and system not only to discuss *information* needed for the task, but also to decide on the *form* of the task itself. As we discuss in Chapter 4, this is a vital feature of supporting agent-based dialogue.

We discuss here finite state models and form-filling models.

2.1.1 Finite State

Finite state dialogue models (e.g., [Hansen, Novick, and Sutton1996]) are the most constrained models we discuss here. In this approach, a dialogue designer must encode not only the domain task, but also *all* possible dialogue sequences. Typically, a finite state automaton is encoded, with each state representing a system utterance (e.g., a prompt) and each transition arc representing a user utterance. Accepting states typically signify successful completion of the domain task (or at least "successful" completion of the dialogue, e.g., the system said goodbye properly).

There are many arguments against finite state models of dialogue (see [Bohlin et al.1999] for some of them). We will only mention a few here which are most relevant to agent-based dialogue.

First, not only do finite state models require a fixed domain task structure (which we will discuss more below with the other models), they also require a fixed, or at least fully-enumerated discourse task structure. This means that a user is restricted to making utterances which correspond to outgoing transitions from the current state, and more generally, must follow one of the predefined paths through the automaton. In practice, this means that the user must talk about task information in the order that the system designer envisioned.

Another shortcoming of finite state models is their inherent lack of memory. The system only has knowledge of what state it is in, but not of how it got there. Long-term dependencies, such as an agreed-upon decision, would be difficult if not impossible to represent in a general way.

2.1.2 Form-filling

In form-filling dialogue (e.g., [Seneff and Polifroni2000; Lamel et al.2000; Chu-Carroll2000]), a frame lists *slots* which represent information needed for the system to perform a (fixed) domain task. Dialogue is modeled as the process of filling in those slots.

This results in a much more free exchange at the discourse-task level, as users can now give information (fill slots) in any order they wish within a single form. Later work [Rudnicky et al.1999; Bohus and Rudnicky2003] added the ability to support the filling of several forms each representing different parts of the domain task. When the form is filled, then the system performs the domain task.

Although form-filling frees one from a fixed discourse task structure, domain task structure remains fixed. A designer must know the domain task structure beforehand in order to design forms that contain the necessary slots for that task.

Arguments against fixed domain task structure are similar to those with fixed discourse task structure mentioned above. Agents often don't have fixed plans for accomplishing goals, and often there may be many possible ways for accomplishing a goal, one of which must be created (planned) by the agent, often on the fly based on the current state of the world. Specifying all possible tasks and their structure beforehand seriously constrains the ability of the agent to adapt to a changing world, and limits the ability of a dialogue system to cover many desirable domains (such as interaction with an autonomous robot).

2.2 Plan-based Models

Plan-based models of dialogue[1] share the view that the purpose of dialogue is to pursue domain goals — either by planning a course of action (joint planning) or by executing such a plan (joint execution). In this section, we first discuss the foundations of plan-based dialogue modeling. We then discuss various threads of research using plan-based dialogue models. Finally, we discuss common shortcomings of these approaches for modeling agent-based dialogue.

2.2.1 Foundations

Austin [1962], Grice [1957; 1969; 1975], and Searle [1975] all noted that human utterances can actually cause *changes* in the world. The utterance "I pronounce you man and wife," said by the right person in the right context actually causes two people to be married. More subtly, the utterance of "John is in the kitchen" may have effect of causing the hearer to believe that John is in the kitchen.

Utterances can have preconditions and effects, the same as other non-linguistic actions. We can also build plans that contain utterances as well as other actions. A full discussion of speech acts is beyond the scope of this thesis. It is important to realize, however, that treating utterances like actions (speech acts) allows us to link it to general theories of planning, execution, and plan recognition in Artificial Intelligence.

Allen, Cohen, and Perrault [1982] were the first to computationalize a theory of speech acts. Cohen [1978][Cohen and Perrault1979] concentrated on using plan synthesis together with speech acts for content planning for language generation. Allen [1979; 1983][Allen and Perrault1980], on the other hand, used plan recognition and speech act theory for intention recognition in language understanding. We concentrate here only on Allen's work.

[1]This section contains material from [Blaylock2002] and [Blaylock, Allen, and Ferguson2003].

Allen studied transcripts of actual interactions at an information booth in a Toronto train station. A typical exchange was something like this [Allen1983]:

Patron: When does the Montreal train leave?
Clerk: 3:15 at gate 7.

Note that although the patron only requested the departure time, the clerk also volunteered information about the departure gate as well. Presumably, the clerk *recognized* the plan of the patron (to board the train), and realized that the patron would also need to know where the train departed and volunteered that information as well. Allen called this behavior *obstacle detection*.

Allen's system took the direct speech act of the utterance, and, using certain inference rules and heuristics to apply them, performed backward chaining in order to infer the user's plan. Heuristics included things such as, if a person wants P, and P is a precondition of action ACT, then the person may want to perform ACT; or if a person wants to know if P is true, they may want P to be true (or false).

Using these inference rules, the system was able to recognize not only indirect speech acts, but also the user's domain plan. Plan-based dialogue models have built upon this foundational work.

Sidner and Israel [1981] extended Allen's work to multiple-utterance dialogues about execution. Their work takes previous discourse context into account in interpreting a new utterance. Each utterance causes the system to update its beliefs about the user's beliefs, wants, goals, and plans, based on Grice's theoretical work [Grice1957; Grice1969] as well as plan recognition similar to Allen's.

2.2.2 Domain-level Plans

Carberry

Carberry [1987; 1990b] models dialogue as joint planning. Her system uses a plan decomposition hierarchy similar to Kautz' [1991] (see Chapter 5) which holds information about decomposition and parameters of plan schemas in the domain. Based on this hierarchy, dialogue utterances build a hierarchical plan by filling in parameters and decomposing subgoals. The system supports planning in both a bottom-up (talk about actions first) and top-down (talk about goals first) fashion.

Although Carberry's system is able to account for dialogues that build complex plans, it only works for utterances that talk directly about actions and goals in the domain. Other dialogue phenomena (such as correction and clarification subdialogues) were not addressed. Also, this system is unable to handle dialogues that support actual plan execution.

Lemon, Gruenstein and Peters

Lemon, Gruenstein and Peters [2002] describe a model similar to that of Carberry, except they model dialogue as joint execution.[2] In their system, dialogue progresses about (possibly) several concurrent execution tasks described as *dialogue threads*. In dialogue, the user can give the system new tasks to execute, state constraints on the execution process, query current execution, abort execution, and so forth. The system not only executes plans, but also uses dialogue to report progress and problems as execution progresses.

In this system, the system and user discuss execution of pre-made plans, but it does not support dialogue for building new plans. Also, coordination of *roles* — or which

[2] In [Lemon, Gruenstein, and Peters2002], it is stated that the model also includes planning, although it is unclear how and to what extent it is supported.

agent performs which action — appear to be hard-coded in plans and not negotiable during dialogue.

2.2.3 Meta-Plans

In order to support dialogue-level phenomena such as correction and clarification subdialogues, several dialogue models also include a meta-plan level.

Litman and Allen

Litman and Allen [1987; 1990][Litman1985; Litman1986] extended Carberry's earlier work to better account for various dialogue phenomena. Although a dialogue's focus is on the domain, there seems to be a meta-layer which helps ensure robust communication.

Essentially, Litman and Allen added a new layer to the dialogue model — meta-plans[3], which are domain-independent plans that take other plans as arguments. Litman and Allen's model is able to account for a number of dialogue phenomena, including that of clarification subdialogues. For example, consider the following dialogue [Litman and Allen1990]:

teacher:	OK the next thing you do is add one egg to the blender, to the shrimp in the blender.
student:	The whole egg?
teacher:	Yeah, the whole egg. Not the shells.
student:	Gotcha. Done.

The domain-level plan here is one of cooking. The student's first utterance ("The whole egg?"), however is uttered because of confusion about the previous utterance. Instead of replying directly to the teacher's instruction, the student asks a clarification

[3] Litman and Allen actually called these *discourse* plans. In light of subsequent work, however, these are better characterized as meta-plans.

question about one of the objects (the egg) to be used in the plan. Litman and Allen model this as an IDENTIFY-PARAMETER meta-plan, as the egg can be seen as a parameter for the plan. The teacher responds to this question and then the student completes the IDENTIFY-PARAMETER plan, and continues with the domain-level plan.

Litman and Allen model meta-plans in a stack-like structure, where new meta-plans can be pushed on the stack in relation to current ones. Other examples of meta-plans include: CORRECT-PLAN (changes a plan when unexpected events happen at runtime), INTRODUCE-PLAN (shifts focus to a new plan), and MODIFY-PLAN (changes some part of a plan).

The addition of meta-level plans allows fuller coverage of various dialogue phenomena, especially correction and clarification subdialogues. While we see this work as an early foundation for our own, we note that the meta-level plan library was somewhat ad hoc and never fully developed. There was also no overall theory of the relations between meta-plans and the over all dialogue strategy of the participants.

Ardissono, Boella and Lesmo

Ardissono, Boella and Lesmo [1996] also introduce a meta-plan, or *problem-solving* plans for decomposing and executing a subgoal.

The problem-solving level consists of two high-level plans: *Satisfy* and *Try-execute* which model a single-agent execution model. In order for an agent to *Try-execute* an action, it first checks constraints and *Satisfies* them if necessary; it then verifies preconditions, does the action and checks the results. Utterances are explained based on some step in this execution model. For example, consider the following dialogue [Ardissono, Boella, and Lesmo1996].

Mark: Sell me a bottle of whiskey.
Lucy: Are you over 18?

Here, Lucy's response can be seen as her performing a *Try-execute* of selling the whiskey. She needs to know if Mark is over 18 in order to verify an execution constraint.

This dialogue model can also be seen as a predecessor of our own, but there are several key differences. First, the model only accounts for execution, but not planning. Also, the single-agent execution model is used for a dialogue participant to recognize the coherence of the other participant's utterance during execution, but it does not directly support *joint* execution, where both dialogue participants coordinate a plan's execution.

Rayner, Hockey and James

Rayner, Hockey and James [2000] don't use meta-plans per se, but rather what they call *meta-output*. In their system, as a user's utterance is interpreted, it is transformed into two signals: a *script*, which is an executable representation the user's command to the system, and *meta-output*, which contains information about the interpretation process. The meta-output is used to report information such as errors and presupposition failures. This meta-output information is used to support dialogue phenomena such as correction and clarification subdialogues.

The dialogue model is again one of execution. The user is essentially giving commands to the system, which the system then executes when it has enough information to do so. The information is necessary to execute a script, however, is computed on the fly, which distinguishes this from a form-filling approach. Similar to the model of Ardissono, Boella, and Lesmo (above), this model only supports dialogue about single-agent execution, and not joint execution, or joint planning.

2.2.4 Other Plan Levels in Dialogue

There have been several efforts to extend the work on meta-plans, specifically that of Litman and Allen, creating other levels of plans to support other dialogue phenomena.

We mention several here.

Lambert

Lambert [1993][Lambert and Carberry1991] proposes a three-level model of dialogue, consisting of the domain and meta levels of Litman and Allen as well as a level of discourse plans, which specifically handles recognition of multi-utterance speech acts. The following utterances, for example, constitute a warning (a discourse plan) only if taken together [Lambert and Carberry1991].

U1: The city of xxx is considering filing for bankruptcy.
U2: One of your mutual funds owns xxx bonds.

The separation of discourse plans allows the recognition of speech-act-like phenomena such as warnings and surprise at a multi-utterance level.

Ramshaw

At the same time as Lambert, Ramshaw [1989; 1989; 1991] proposed a differently separated three-level model. Instead of a discourse level, Ramshaw proposed an *exploration* level. The intuition is that some utterances are made simply in the attempt to *explore* a possible course of action, whereas others are explicitly made to attempt to *execute* a plan.

Ramshaw's model allows the system to distinguish between the two cases, He uses a stack-based approach, with exploration-level plans pushed on lower-level plans. Unfortunately, this approach means that exploration-level and domain-level plans must be separate. Once one starts exploring, one cannot talk about the plan-execution level until done exploring. As observed in [Carberry, Kazi, and Lambert1992], this prevents understanding of contingent commitments when one is at the exploration level. One cannot build two complex, competing plans and compare them. Also, the model could

not handle bottom-up dialogues (dialogues which start with atomic actions and build their way up).

Although Ramshaw's model allows for some problem-solving behavior, including comparisons of different plans, it does not model this collaboratively. Instead, individual comparisons can only be detected, but they do not affect the planning state.

Carberry, Kazi and Lambert

Carberry, Kazi an Lambert [1992] incorporate Ramshaw's work into the previous work by Lambert (see above). They overcome the problem with contingent commitments and also modeling the building and comparing of complex plans. Their model also allows for the handling of both top-down (goal first) and bottom-up (actions first) dialogues.

However, the problem-solving level was fairly undeveloped, mentioning only exploring different recipes, choosing the best, and doing the same at lower levels. There are also several other shortcomings of this approach.

First, the plan structure requires that these meta plans be executed linearly. In other words, one must first explore all recipes, and once this is fully done, choose the best, etc. The model does not appear to support the revisiting of previous decisions by an agent.

Second, there is no execution in the model. It only provides for dialogues about planning. Also, although there is a limited problem-solving model, there is no notion of collaboration. It is not clear how each participant could contribute separately to the plan being built. Sample dialogues are all of the master-slave type [Grosz and Sidner1990], where one participant is proactive in planning and the other basically serves as an information source.

Chu-Carroll and Carberry

Chu-Carroll and Carberry [1994; 1995; 1996; 2000] extend the three-level model of Lambert and add a fourth level, belief. They also changed the model so that it distinguished between proposed and accepted plans and beliefs. This extends coverage to include *negotiation dialogues*, where participants have conflicting views and collaborate to resolve them. The model is based on Lambert's work, and shares the same shortcomings as we mentioned above.

2.2.5 SharedPlans

Another thread of plan-based dialogue modeling has been in the SharedPlans approach. One of the shortcomings of many of the plan-based systems mentioned above is that, although they model dialogue on plans, they do not explicitly model the collaborative nature of dialogue.

The SharedPlan formalism [Grosz and Kraus1996; Grosz and Kraus1999] was created in part to explain the intentional structure of discourse [Grosz and Sidner1986; Grosz and Sidner1990; Lochbaum, Grosz, and Sidner2000]. It describes how agents collaborate together to form a joint plan. The model has four operators which are used by agents in building SharedPlans.

- *Select_Rec*: An individual agent selects a recipe to be used to attain a given subgoal.
- *Elaborate_Individual*: An individual agent decomposes a recipe into (eventually) completely specified atomic actions.
- *Select_Rec_GR*: Intuitively, the same as Select_Rec, only at the multi-agent level.[4] A group of agents select a recipe for a subgoal.

[4] Individual and group operators entail different constraints on individual intentions and beliefs. However, this is not important for understanding the formalism as a model of collaborative planning.

- *Elaborate_Group*: The multi-agent equivalent of Elaborate_Individual — a group of agents decompose a recipe.

Using these four operators, a group of agents collaborates until it has completely specified a *full SharedPlan* (which they will presumably execute at some time in the future).

Based on the theoretical formalism, Lochbaum [1998] developed an intention recognition algorithm that works on the process of *plan augmentation*. In her algorithm, an utterance causes the hearer to ascribe certain intentions and beliefs to the speaker. If willing, the hearer also adopts those intentions and beliefs. As a result of the new beliefs and intentions, the SharedPlan is augmented, i.e., brought one step closer to completion.

At a more concrete level, the algorithm attempts to segment dialogue into a stack of *discourse segments*, which roughly correspond to the SharedPlan operators mentioned above. At each new utterance, the algorithm decides if it is (a) completing a discourse segment, (b) continuing the current discourse segment, (c) pushing a new discourse segment on to the stack, or (d) some combination of these functions. The algorithm was specified at a very high level, and appears to have never been fully implemented.

The main focus of the SharedPlan model has been to formalize agent intentions and beliefs in forming and sharing joint plans, something which is weak in our model. However, for our purposes — supporting agent-based dialogue — there are several shortcomings in the SharedPlans model.

First, SharedPlans only models collaboration for joint planning between agents. It does not model the collaboration that occurs when agents are trying to *execute* a joint plan.[5]

---

[5] Although the formalism does specify the needed intentions and beliefs for agents executing joint plans.

Second, the SharedPlans formalism models the formulation of joint plans with the four operators previously discussed: *Select_Rec*, *Elaborate_Individual*, *Select_Rec_GR*, and *Elaborate_Group*. Although these operators were sufficient to allow the formalization of group intentions and beliefs about joint plans, they do not provide enough detail for us to model collaboration at an utterance-by-utterance level (which is needed, among other things, to represent communicative intentions). As an example, consider the *Elaborate_Group* operator, which has the function of decomposing a recipe, instantiating the parameters (including which agent or subgroup will perform which action at what time and which resources will be used), and making sure the rest of the group has similar intentions and beliefs about the plan. An *Elaborate_Group* can (and often does) consist of many individual utterances. In order to build a dialogue system, we need to be able to model the communicative intentions behind a single utterance.

We actually believe that our model may be compatible with the SharedPlans formalism and can be seen as specifying the details of the SharedPlan operators.[6]

COLLAGEN

COLLAGEN [Rich and Sidner1998; Rich, Sidner, and Lesh2001] is a general toolkit for building collaborative interface systems (natural language or otherwise) in which user interaction with an application is modeled with a subset of the SharedPlans model. In order to "port" COLLAGEN to a new application, a developer needs just to model domain plans and provide an agent to interact with both COLLAGEN and the application to be interfaced. The actual interaction modeling and management is handled automatically by COLLAGEN, thus simplifying the developer's task as well as providing consistency across applications.

The COLLAGEN interaction manager models dialogue using an implementation of Grosz and Sidner's tripartite structure of dialogue [Grosz and Sidner1986]. Utterances

[6]This is actually mentioned in [Grosz and Kraus1996] as an area of needed future work.

(and other actions) are grouped into segments (linguistic level), which have pointers into a plan-tree structure (intentional level), as well as a focus stack (attentional level).

Utterances themselves are modeled with a subset of Sidner's artificial negotiation language [Sidner1994; Sidner1994], although to our knowledge, the content language was never fully specified. The most common examples in publications have had *ProposeForAccept* (*PFA*) and *AcceptProposal* (*AP*) at a top level, with *SHOULD* and *RECIPE* nested within them. The action *PFA(RECIPE)* means that the agent (artificial or human) is proposing to use the given recipe. The action *PFA(SHOULD)* seems to roughly correspond to the agent (or human) either volunteering to perform an action, or assigning the other agent to perform it. However, whether this proposal is to add the action to the plan or to actually begin execution seems ambiguous. Examples in the literature have only dealt with execution and not planning. COLLAGEN uses an implementation of the principles in Lochbaum's intention recognition algorithm [Lochbaum1998] to update discourse structures based on how the newly observed action/utterance fits into the understood recipe library.

Using the operators above, COLLAGEN seems[7] to cover dialogue about goal selection and plan execution (including hierarchical decomposition of predefined recipes). However, as the intentional structure does not distinguish between whether a node was only planned or actually executed, it does not appear that COLLAGEN covers dialogue supporting both planning and execution. Also, although it allows for mixed-initiative collaboration through the use of proposals and acceptance/rejection, it does not appear to include a mechanism of revising previous decisions (e.g., in replanning).

[7]Again, it is in some ways difficult to make an assessment of COLLAGEN's representational capabilities, as the utterance representation language was never fully specified.

2.2.6 Rational-Behavior Models

Several researchers [Cohen and Levesque1990c; Cohen et al.1991; Cohen1994; Sadek and De Mori1998] have suggested that dialogue should simply be modeled as *rational behavior* of agents (cf. [Cohen and Levesque1990a; Cohen and Levesque1990b; Levesque, Cohen, and Nunes1990]). In a nutshell, these models predict that agents communicate because they are committed to by the principles of rational behavior.

As an example, agents, in order to achieve goals, form joint intentions. These joint intentions commit rational agents to certain behavior such as helping their fellow agents achieve their part of the plan and letting the other agents know if, for example, they believe the goal is achieved, or they decide to drop their intention. These commitments and rationality can be seen as what causes agents to engage in dialogue.

On the surface, these models may seem to be what is needed to support agent-based dialogue, since they are modeled as agent collaboration. However, we see several shortcomings in these models.

First and foremost, rational behavior models have only modeled dialogue supporting joint execution of plans and do not handle dialogue that supports joint planning.[8]

Also, although the single-agent levels of formal representations of individual and joint agent rationality are very thorough, the levels of dialogue modeling and agent interaction were never fully developed as far as we are aware.

2.2.7 General Shortcomings of the Plan-based Approach

Here we mention two of the general shortcomings in the previously-mentioned systems, which motivate our own work: the limitations of their problem-solving models and

[8]This likely stems from the fact that the theoretical models of rationality upon which these systems are based (e.g., [Cohen and Levesque1990a]) focus on formalizing agent execution, and not planning. In a way, work on rational behavior models can be seen as a kind of complement to work on SharedPlans — which focuses on planning, but not execution.

collaboration paradigms.

Problem-Solving Models As discussed in Chapter 1, we would like to model a wide range of dialogue, especially dialogue which supports different types of tasks. Most previous plan-based dialogue models have only supported dialogue which supports planning *or* dialogue which supports execution, but not both. However, as the example dialogue in Figure 1.1 illustrates, dialogue can often be used to support both planning and execution. In fact, dialogue can support just about any facet of collaborative activity. This includes not only just planning and execution, but also goal selection, plan evaluation, execution monitoring, replanning, and a host of others. In addition, conditions often change, requiring dialogue participants to go back and revisit previous decisions, reevaluate and possibly even abort goals and plans. Previous models have sometimes handled a few of these activities, but none, as far as we are aware, has handled them all.

Collaboration Paradigms A *collaboration paradigm* describes the respective roles and authority each participant has during collaboration. Participants may, for example, have different social statuses, giving rise to different collaboration paradigms. If two participants are on equal social footing, then they may both be free to make and reject proposals as they see fit. Decisions are discussed and made jointly. This paradigm is often referred to as *mixed-initiative* [Chu-Caroll and Brown1997].

At the other extreme is the *master-slave* collaboration paradigm [Grosz and Sidner1990], in which one participant completely controls the flow of the collaboration as well as the decisions made. There is a whole spectrum of collaboration paradigms between the extremes of mixed-initiative and master-slave. In a *boss-worker* paradigm, for example, the socially higher (boss) participant likely controls most of the discussion, but the worker participant may be expected to make contributions at certain levels but not be allowed to disagree with the boss.

With the exception of [Chu-Carroll and Carberry2000], previous work in intention recognition has only modeled master-slave collaboration. Most previous research was restricted to information-seeking dialogues (e.g., [Carberry1990b]), where collaboration consists of the user (the master) getting information from the system (the slave). Although the system may possibly ask clarification questions, it cannot take task-level initiative [Chu-Caroll and Brown1997] and is not party to the user's planning decisions. Expert-apprentice dialogues (e.g., [Grosz1981]) also fit this model. In these, the expert is the master, and the apprentice only follows the plan the expert puts forth.

This master-slave assumption limits the types of collaboration which can be modeled. We are interested in modeling the entire spectrum of collaboration paradigms, from master-slave to mixed-initiative.

3 A Model of Collaborative Problem Solving

As we discussed in Chapter 2, current dialogue models are unable to support the kind of agent-based dialogue that we are interested in. This is mainly due to two shortcomings: (1) the models cover dialogue about very narrow set of agent behavior — typically either planning or execution; and (2) most models do not support the range of collaboration paradigms, especially mixed-initiative — where both parties have an equal say in the decision-making.

In this chapter, we present a model of agent-agent collaborative problem solving which will serve as the foundation of our agent-based dialogue model presented in Chapter 4. The collaborative problem-solving model serves as a model of agent-agent communication which takes into account a range of agent behavior and allows for a range of collaborative paradigms. This then allows us in the next chapter to build an agent-based dialogue model that overcomes the two shortcomings described above.

Although much work has been done on language specification for (artificial) agent communication, most (e.g., [DARPA Knowledge Sharing Initiative, External Interfaces Working Group1993]) has focused on the meaning of individual messages or *utterances* and not on how those utterances contribute to ongoing collaboration between the agents.

Because they only deal with collaboration among artificial agents, most multi-agent systems define and use ad hoc interaction protocols (often finite-state based as in [The

Foundation for Intelligent Physical Agents2002]) which allow agents to understand utterances in the context of the current "dialogue". As discussed in Chapter 2, such fixed-dialogue representations are only able to account for a small range of human dialogue behavior.

On the other hand, work on general agent collaboration (e.g., [Cohen et al.1991; Grosz and Kraus1996; Wooldridge and Jennings1999]) typically formalizes collaboration in a logic of individual agent mental state. However, the processes described are too high-level to describe interaction at the granularity of human utterances (as described in Chapter 2).

In this chapter, we present a model of collaborative problem solving which represents a kind of "middle ground" of the work mentioned above. We model collaboration at a level suitable to represent individual utterances without placing restrictions on the form of collaboration itself. Although we do not provide a formal model in terms of individual agent mental state, we believe our model may be compatible with other work on general agent collaboration. In Section 3.4.4, we describe possible compatibility with the SharedPlans model [Grosz and Kraus1996]. We also note that we make the simplifying assumption here, that there are only two agents collaborating. We are hopeful, however, that the model discussed here can be extended to the general case of many agents. We leave this as an area of future research.

In the remainder of the chapter, we first provide an intuitive description of what we mean by problem solving and collaborative problem solving. We then describe the collaborative problem-solving model and its applicability as a general model of agent-agent communication.

3.1 The Collaborative Problem-Solving Process

Before we describe our model, it is first important to describe what we mean by problem solving and collaborative problem solving, as these terms are widely used but seldom

defined in the literature. We first describe (single-agent) problem solving and then extend the description to collaborative problem solving between two agents.

3.1.1 Single-Agent Problem Solving

We define problem solving (PS) to be the process by which an agent chooses and pursues goals or *objectives*. Specifically, we model it as consisting of the following three general phases:

- *Determining Objectives*: In this phase an agent manages objectives, deciding to which it is committed, which will drive its current behavior, etc.
- *Determining and Instantiating Recipes for Objectives*: In this phase, an agent determines and instantiates a recipe to use to work towards an objective. An agent may either choose a recipe from its recipe library, or it may choose to *create* a new recipe via planning.[1]
- *Executing Recipes and Monitoring Success*: In this phase, an agent executes a recipe and monitors the process to check for success.

There are several things to note about this general description. First, we do not impose any strict ordering on the phases above. For example, an agent may begin executing a partially-instantiated recipe and do more instantiation later as necessary. An agent may also adopt and pursue an objective in order to help it in deciding what recipe to use for another objective. In the section below, we provide several examples of behavior that we consider to be problem solving.

It is also important to note that our purpose here is not to specify a specific *problem-solving strategy* or prescriptive model of how an agent *should* perform problem solving. Instead, we want to provide a general descriptive model that encompasses the many

[1] Actually, both of these activities (instantiating a recipe versus creating a new one) have been called planning in the literature.

possible problem-solving strategies agents may have. For example, an agent which simply executes recipes and reasons little about them will be reactive but brittle, whereas an agent that constantly re-evaluates its objectives and recipes will be flexible, but slow. This reflects the variance of problem-solving strategies in humans: some of us are cautious and slower to react, while others make decisions only once and then stick to them. In fact, an individual may use different strategies in different situations. In the human world, people with problem-solving strategies from one extreme can still collaborate with people from the other extreme. Our model needs to allow for collaboration among heterogeneous agents as well.

Examples of Problem-Solving Behavior

In order to better illustrate the possible range of problem solving, we give several simple examples of problem-solving behavior. This is not meant to be an exhaustive list.

- *Prototypical*: Agent Q decides to go to the park (objective). It decides to take the 10:00 bus (recipe). It goes to the bus stop, gets on the bus and then gets off at the park (execution). It notices that it has accomplished its objective, and stops pursuing it (monitoring).

- *Subordinate Objective*: Agent Q decides to go to the park (objective1). In order to decide which recipe to use, it decides to see what the weather is like (objective2) by looking outside (recipe for objective2). It goes to the window and looks outside (execution) and notices that it is sunny. It decides to walk to the park (recipe for objective1)....

- *Interleaved Planning and Execution*: Agent Q decides to to go to the park. It decides to take a bus (partial recipe) and starts walking to the bus stop (partial execution) as it decides which bus it should take (continues to instantiate recipe)....

- *Replanning*: Agent Q decides to go to the park. It decides to walk (objective) and goes outside of the house (begins execution). It notices that it is raining and that it can't successfully walk to the park[2] (monitoring). It decides instead to take the 10:00 bus (replanning)....

- *Abandoning Objective*: Agent Q decides to go to the park by taking the 10:00 bus. As it walks outside, it notices that it is snowing and decides it doesn't want to go to the park (abandons objective). It decides to watch TV instead (new objective)....

3.1.2 Collaborative Problem Solving

Collaborative problem solving (CPS) follows a similar process to single-agent problem solving. Here two agents jointly choose and pursue objectives in the same stages (listed above) as single agents.

There are several things to note here. First, the level of collaboration in the problem solving may vary greatly. In some cases, for example, the collaboration may be primarily in the planning phase, but one agent will actually execute the plan alone. In other cases, the collaboration may be active in all stages, including the planning and execution of a joint plan, where both agents execute actions in a coordinated fashion. Again, we want a model that will cover the range of possible levels of collaboration.

Another thing to note is that, as discussed in Chapter 2, we need to be able to handle the range of collaboration paradigms (the respective roles and authorities of each agent) — from master-slave to mixed-initiative. In some cases, for example, one agent may have the authority to make all decisions while the other may just provide suggestions (a kind of boss-worker paradigm). In others, both agents may be totally autonomous and need to negotiate to agree on decisions.

---

[2] At least, it can't walk there and still maintain its objective to stay dry (cf. [Wilensky1983]).

3.2 A Collaborative Problem-Solving Model

We now describe our collaborative problem-solving (CPS) model. We will first give a brief overview of the model and then describe the individual parts in detail.

We believe that the general collaborative problem-solving process remains the same, regardless of the task and domain (cf. [Allen et al.2000; Allen et al.2001]). Thus the CPS model is built to be domain-independent. *Task models* are used as a type of "plug-in" to specialize the model to a particular domain. Objects in the task model are specializations of abstract objects in the (single-agent and collaborative) problem-solving models. The PS and CPS models are comprised of a set of acts whose execution updates the (single-agent or collaborative) *problem-solving state*. Acts at this level include such things as evaluating and adopting objectives and recipes, executing plans, and so forth.

At the CPS level, it is impossible for an agent to single-handedly change the collaborative problem-solving state. Doing so involves the cooperation of both agents involved (cf. [Traum1994]). This means that CPS acts are *not* directly executable by a single agents. How is a single agent to affect the CPS state, then? It is done through negotiation with the other collaborating agent.

Interaction acts are single-agent actions used to negotiate changes in the CPS state. If the agents are amenable to the change and cooperate, a certain combination of interaction acts will result in the generation of a CPS act, changing the CPS state. In collaborative problem solving between artificial agents, these interaction acts can be used directly in a communication language. However, in natural language, they must be encoded in (and decoded from) (natural language) *communicative acts*. (This connection to natural language is actually outside of the CPS model proper, and will be discussed in more detail in Chapter 4.)

At this point, we describe a single-agent model of problem solving. After that, we show how it is extended to the collaborative case.

3.3 Single-agent Problem Solving

This section describes a model of single-agent problem solving. We should note from the outset that the single-agent problem solving model described here is not meant to compete with other, well-known models of mental states for autonomous agents (e.g., [Cohen and Levesque1990a; Rao and Georgeff1991]). We are not proposing that this be the basis of an implementation of an actual agent. Rather, we present this as a kind of abstraction of single-agent mental models. We do this for two reasons: first and foremost, it is instructive to look at single-agent problem-solving behavior to compare and contrast what is done in collaboration. Second, although not mentioned further in this thesis, it is important for collaborating agents to have a model of the other agent's current mental state (cf. [Cohen and Levesque1990c; Grosz and Kraus1996]). We believe that the model presented here may be at the right level of abstraction to provide such other-agent modeling, although we must leave this as a topic of future research.

Central to the single-agent model is the *problem-solving state*, which is an agent's mental model of its current state of the problem-solving process. The PS state is composed of a number of *problem-solving objects* and their status within the problem-solving process. An agent can change its PS state by executing certain mental operators called *problem-solving acts*.

In this section, we first discuss PS objects, then how they combine to form the PS state. We then discuss how an agent can use PS acts to change the PS state.

3.3.1 PS Objects

The basic building blocks of the PS state are PS objects, which we represent as typed feature-value structures. PS object types form a single-inheritance hierarchy, where children inherit or specialize features from parents. Instances of these types are then used in problem solving.

In our PS model, we define types for the upper level of an ontology of PS objects, which we term *abstract PS objects*. These abstract PS objects are used to model problem-solving at a domain-independent level.

We first explain in more detail the representation of PS objects, and afterwards define and explain each of the abstract PS objects. We then discuss how these objects can be specialized to model a particular domain.

Representation of PS Objects

We represent all objects (PS as well as other auxiliary objects) as typed feature structures. For our own representation and the explanation here, we borrow heavily from [Pollard and Sag1994], although we use terminology more familiar in the planning field.

Types An object type[3] declares which features an instance of that type must have, as well as the allowable types for the values of those features. Formally, an object type declaration is of the following form:

$$\begin{array}{c} \sigma \leftarrow \rho \\ \begin{bmatrix} F_1 & \tau_1 \\ \vdots & \vdots \\ F_n & \tau_n \end{bmatrix} \end{array}$$

where $\sigma, \rho, \tau_1, \ldots, \tau_n$ are types and $F_1, \ldots, F_n$ are feature labels. Here σ is the type currently being defined and ρ is its immediate parent in the inheritance hierarchy. $\tau_1, \ldots, \tau_n$ define the allowed type of value for each corresponding feature label $F_1, \ldots, F_n$.

A type inherits all of its parent's feature labels and their corresponding type restrictions. A type may change a type restriction from an inherited feature label F from τ_1

[3] [Pollard and Sag1994] use the term *sort* instead of type.

$$\textbf{object} \leftarrow \epsilon$$

$$\begin{bmatrix} \text{ID} & \textit{id} \end{bmatrix}$$

Figure 3.1: Type Description for *object*

to τ_2 iff τ_2 is a descendant of τ_1. Usually, we will not list unchanged, inherited features in new type declarations, although we do occasionally when it makes the explanation more clear.

In order to ensure that all objects in our problem-solving model are labeled with a unique ID, we introduce in Figure 3.1 a basic type *object* which has a single attribute ID.[4] We then require that all objects in our hierarchy be descendants of *object*.

We use the following atomic types, which are not defined here, but are taken to have their typical meaning: *string* and *number*. We also use the atomic type *id*, which we do not define, but it is something which allows each object to receive a unique ID.

We define several type templates: *set(σ)*, *list(σ)*, and *stack(σ)*, where the set, list or stack is restricted to elements of type σ. When displaying a set in a token, we will use curly brackets ({}); for a list, we use we use angle brackets (<>); and for a stack, we use parens (()).

Finally, atomic types may also be defined by enumerating a finite set of elements (like an `enum` in C). For the abstract model, we define the enumeration *boolean* = {*true*,*false*}.

Tokens Feature-value tokens can be fully or partially instantiated,[5] and we will display them in this text as attribute-value matrices (AVMs). Following [Pollard and Sag1994], we also use paths of the form A | B | C to easily refer to embedded content

[4]Note that the line ***object*** $\leftarrow \epsilon$ signifies that *object* is a root in the hierarchy.

[5][Pollard and Sag1994] require a token to be fully instantiated.

(in this case, the contents of the C feature embedded in the B feature of A). We also use boxed numbers (e.g., $\boxed{1}$) to refer to entire tokens and to signify structure sharing.

An example of an instantiated token is shown later in Figure 3.28.

Abstract PS Objects

The following are the six abstract PS objects from which all other domain-specific PS objects inherit:

Objective A goal, subgoal or action. For example, in a rescue domain, objectives could include rescuing a person, evacuating a city, and so forth. We consider objectives to be actions rather than states, allowing us to unify the concepts of action and goal.

Recipe An agent's beliefs of how to attain an objective. Although we do not adhere to any specialized definition of recipe, one example is Carberry's domain plan library [Carberry1990b] which has action decomposition information about objectives. An agent's recipe library can be expanded or modified through (collaborative or single-agent) planning.

Constraint A restriction on an object. Constraints are used to restrict possible solutions in the problem-solving process as well as possible referents in object identification.

Evaluation An agent's assessment of an object's value within a certain problem-solving context. Agents will often evaluate several competing possible solutions before choosing one.

Situation The state of the world (or a possible world). In all but the simplest domains, an agent may only have partial knowledge about a given situation.

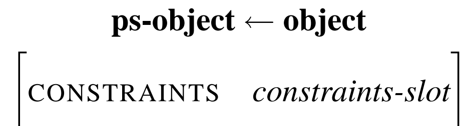


Figure 3.2: Type Description for *ps-object*

Resource All other objects in the domain. These include include real-world objects (airplanes, ambulances, ...) as well as concepts (song titles, artist names, ...)

Each of the abstract PS objects share a set of common features. We put these common features in a new type, *ps-object*, which is the common parent of all of the abstract PS objects. The type description for *ps-object* is shown in Figure 3.2. We briefly describe its features here and then continue by giving the type declarations for each of the abstract PS objects in turn.

ps-object inherits from *object* and therefore contains an ID attribute (not shown in the figure). It also has one additional attribute: CONSTRAINTS. The CONSTRAINTS attribute provides a way of describing the *ps-object* with a set of *constraints*, which may not be particularly useful for single-agent problem solving, but which is frequently used in a collaborative setting, when agents try to refer to the same object. This is described in more detail when we introduce the *constraint* type below.

It is important to note that the type of the CONSTRAINTS attribute is not simply a set of type *constraint*. Rather, it is one of a special class of middleman types we call *slots*. As slots are a vital part of the PS model, we take a brief aside here to discuss them before continuing with the abstract PS objects.

Slots and Fillers Problem solving can be seen as an agent's decision-making process with respect to choosing and pursuing objectives. In modeling problem solving, we need to model more than just the decisions made. We need to model the decision-making *process* itself.

Within our model, decisions can be seen as the choosing of values (objects) or sets of values for certain roles. For example, an agent decides on a set of objectives to pursue. For each objective it has, an agent must decide on a (single) recipe to use in pursuing it, and so forth. A straightforward way of modeling these decisions would be to include, for example, a feature RECIPE in an objective which takes a recipe value, and represents the current recipe the agent is using to pursue this objective. Similarly, we could define a feature OBJECTIVES at the top level which would hold the set of objectives the agent is currently committed to.

However, doing this would only model the agent's *decision*, but not the *process* the agent followed in making that decision. In deciding on a recipe to use for an objective, an agent may identify several possible recipes as possibilities and evaluate each one. It may similarly narrow down the space of possible recipes by placing constraints on what it is willing to consider. These kinds of meta-decisions are rarely explicitly modeled in agents, and it may seem like overkill to do so. However, as we will see below, these kind of meta-decisions are what a large bulk of collaborative communication is used for!

To be able to model these and other kinds of decisions-making processes, we add two levels of indirection at each decision point in the model. The first is what we call a *slot*, which contains information about the possible filler values (e.g., recipes) which have been/are under consideration in that context. A slot also contains information about possible constraints which have been put on what should be considered (e.g., not *all* valid recipes, but just those which take less than 30 minutes to execute). A slot also records which (if any) *filler* has been chosen by the agent.

A *filler* is the second layer of indirection. It is used to wrap an actual value with a set of evaluations the agent has made/might make about it. Note that this wrapping is necessary, as evaluations will always be context-dependent (i.e., dependent on the current slot) and not attached in general to the value itself.

Using these two levels of indirection, slots and fillers, gives us a rich model of not

slot ← **object**

$$\begin{bmatrix} \text{IDENTIFIED} & \textit{set(filler(ps-object))} \end{bmatrix}$$

Figure 3.3: Type Description for *slot*

single-slot ← **slot**

$$\begin{bmatrix} \text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler(ps-object))} \\ \text{ADOPTED} & \textit{filler(ps-object)} \end{bmatrix}$$

Figure 3.4: Type Description for *single-slot*

only the decisions (to be) made, but also the decision-making process itself.

In Figure 3.3, we define an abstract *slot* type, which is the parent of *single-slot* (Figure 3.4) and *multiple-slot* (Figure 3.8). These types differentiate decision points where just one filler is needed (e.g., a single recipe for an objective), or where a set of values can be chosen (e.g., objectives that the agent wishes to pursue). We first discuss the single case, and then the multiple case.

Slots for Single Values The most typical case is where a single value can be used to fill a slot. *single-slot* is an abstract class for handling this. It has three features (besides the ID inherited from *object*): IDENTIFIED is the set of all values (wrapped in *fillers*) that an agent has considered/is considering to fill this slot. ADOPTED records the single value which the agent has committed to for this slot. (This value may also be empty in the case that the agent has not yet decided, or has reversed a previous decision.) The CONSTRAINTS feature describes possible constraints the agent has put on possible slot fillers (such that the chosen recipe have 5 or fewer steps). Note that this is itself a type of slot, *constraints-slot*, which inherits from *multiple-slot* as it can contain a set of

$$\textbf{filler} \leftarrow \textbf{object}$$
$$\begin{bmatrix} \text{EVALUATIONS} & \textit{evaluations-slot} \\ \text{VALUE} & \textit{ps-object} \end{bmatrix}$$

Figure 3.5: Type Description for *filler*

$$\textbf{filler}(\sigma) \leftarrow \textbf{filler}$$
$$\begin{bmatrix} \text{EVALUATIONS} & \textit{evaluations-slot} \\ \text{VALUE} & \sigma \end{bmatrix}$$

Figure 3.6: Type Description for *filler(σ)*

constraints. We will describe the *constraints-slot* type shortly.

Note also that the types of the IDENTIFIED and ADOPTED features contain a *filler* type. Figure 3.5 shows the base definition of a *filler*. It contains both a VALUE which it wraps, as well an EVALUATIONS attribute, which represents any evaluations the agent may make/have made about the value in the local context. (The EVALUATIONS attribute is also a type of slot which inherits from *multiple-slot* which we will also come to shortly.)

The abstract *filler* type in Figure 3.5 only restricts its VALUE to be a *ps-object*. In most cases, we want to restrict this further to be the type of the expected value (e.g., a recipe). In Figure 3.6, we define a type schema *filler(σ)* which allows us to easily refer to a subtype of *filler* which specializes the VALUE to type σ.[6]

Similarly, we usually need to specify a *single-slot* to allow only a certain type of filler. Figure 3.7 defines a type schema which, similar to what we did with *filler* above,

[6]For reasons of clarity, we have also used this notation in the definition of *single-slot* in Figure 3.4, even though σ is *ps-object* here and therefore the instantiated schema simply resolves to the abstract type *filler*.

$$\textbf{single-slot}(\sigma) \leftarrow \textbf{single-slot}$$
$$\begin{bmatrix} \text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler}(\sigma)\textit{)} \\ \text{ADOPTED} & \textit{filler}(\sigma) \end{bmatrix}$$

Figure 3.7: Type Description for *single-slot(σ)*

$$\textbf{multiple-slot} \leftarrow \textbf{slot}$$
$$\begin{bmatrix} \text{IDENTIFIED} & \textit{set(filler(ps-object))} \\ \text{ADOPTED} & \textit{set(filler(ps-object))} \end{bmatrix}$$

Figure 3.8: Type Description for *multiple-slot*

refers to a subtype of *single-slot* which only allows identified and adopted values of type *filler(σ)*.

Slots for Sets of Values Figure 3.8 defines an abstract slot for decisions which allow more than one simultaneous value. In our model, we use three classes which inherit from *multiple-slot*: *constraints-slot*,*evaluations-slot*, and *objectives-slot*. We describe each in turn.

Previously-discussed types *ps-object* and *single-slot* have already introduced the *constraints-slot* type. Its definition is shown in Figure 3.9. As discussed above, this

$$\textbf{constraints-slot} \leftarrow \textbf{multiple-slot}$$
$$\begin{bmatrix} \text{IDENTIFIED} & \textit{set(filler(constraint))} \\ \text{ADOPTED} & \textit{set(filler(constraint))} \end{bmatrix}$$

Figure 3.9: Type Description for *constraints-slot*

evaluations-slot ← multiple-slot

$$\begin{bmatrix} \text{CONSTRAINTS} & \textit{constraints-slot} \\ \text{IDENTIFIED} & \textit{set(filler(evaluation))} \\ \text{ADOPTED} & \textit{set(filler(evaluation))} \end{bmatrix}$$

Figure 3.10: Type Description for *evaluations-slot*

type allows a set of constraints to be identified and adopted in a context. Here IDENTIFIED and ADOPTED have a similar meaning to those in *single-slot*, with the only exception being that ADOPTED takes a set of *fillers*, instead of a single value. We discuss constraints in more detail when we come to their definition within the PS model below.

Note that, unlike all the other PS slot types, *constraints-slot* does not contain a CONSTRAINTS attribute. Theoretically, we believe it is possible for an agent to set constraints on which constraints it would consider adopting, but practically, nesting a *constraints-slot* here would create an infinite regress of constraints-slots. We have therefore chosen instead to exclude this from the model as it stands, and leave it as a subject of future research.

The second slot for multiple values is the *evaluations-slot*, which we used above in the definition of *filler*. An *evaluation-slot* (defined in Figure 3.10) provides a space for determining a set of *evaluations*. Its attributes are used in the same way to those of *single-slot* and *constraints-slot*, and do not merit further comment here.

The final slot type for multiple values is the *objectives-slot*, which is defined in Figure 3.11. It too has the features CONSTRAINTS, IDENTIFIED and ADOPTED which are used as they are in *evaluations-slot*. The reason that objectives use a multiple slot and not a single slot will be discussed below in the various abstract PS objects where the *objectives-slot* is used. For now, it is just important to understand that an objective being adopted means that the agent is committed to that objective within the local PS

objectives-slot ← multiple-slot

CONSTRAINTS	*constraints-slot*
IDENTIFIED	*set(filler(objective))*
ADOPTED	*set(filler(objective))*
SELECTED	*set(filler(objective))*
RELEASED	*set(filler(objective))*

Figure 3.11: Type Description for *objectives-slot*

objective ← ps-object

RECIPE	*single-slot(recipe)*

Figure 3.12: Type Description for *objective*

context.

Objectives are not only committed to, but can also be executed. Objectives in the SELECTED set are those which the agent is currently executing (more details below), as opposed to just intending to execute.

Finally, as discussed above, an agent must monitor the situation in order to notice when an objective has been fulfilled (so that it stops pursuing it). Objectives which the agent believes have been fulfilled are put into the RELEASED set.

Objective Now that we have described the various slot and filler types which are used in the model, we are ready to get on with the definitions of the abstract PS object types. The definition of *objective* is shown in Figure 3.12. This object, like all six abstract PS objects, inherits directly from *ps-object* and thus contains its attributes (shown in Figure 3.2). In addition, *objective* has a RECIPE attribute which is of type *single-slot(recipe)*. This slot provides a place to record all problem-solving activity

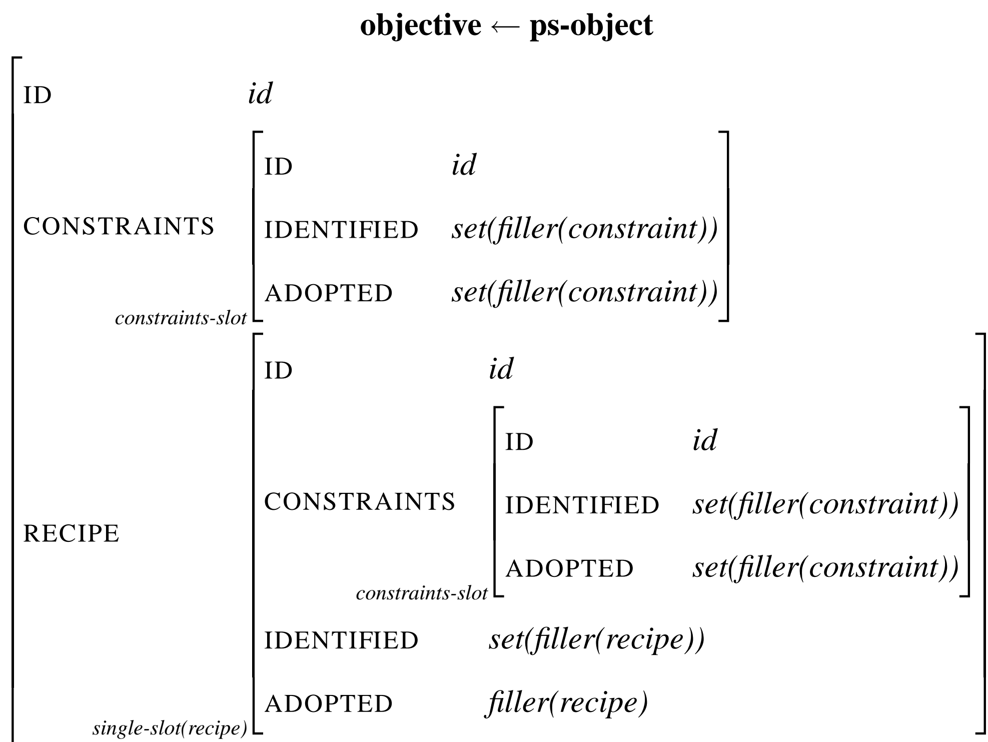


Figure 3.13: Full Type Description for *objective*

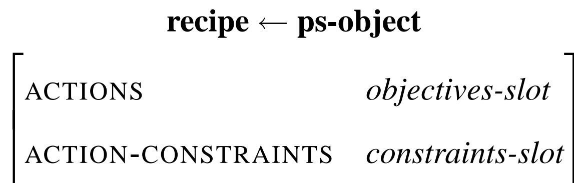


Figure 3.14: Type Description for *recipe*

related to choosing a single *recipe* to use to pursue the *objective*.

The fully-expanded *objective* type description is shown in Figure 3.13. This includes features inherited from *ps-object* (and indirectly from *object*) as well as the result of expanding the *single-slot* and *constraints-slot* templates.

constraint ← **ps-object**

$$\begin{bmatrix} \text{EXPRESSION} & \textit{boolean-expression} \end{bmatrix}$$

Figure 3.15: Type Description for *constraint*

resource ← **ps-object**

$$\begin{bmatrix} \text{ACTUAL-OBJECT} & \textit{id} \end{bmatrix}$$

Figure 3.16: Type Description for *resource*

Recipe Recipes are represented as a set of subobjectives (actions) and a set of constraints on those subobjectives. The attributes of *recipe* are shown in Figure 3.14. The ACTIONS attribute is a *objectives-slot* which allows a set of *objectives* associated with the *recipe*, as discussed above. The attribute ACTION-CONSTRAINTS contains the *constraints* placed on the *objectives*.

Constraint *Constraints* (Figure 3.15) are represented as *boolean-expressions*. We do not define the form of these expressions here, but we envision a typical kind of expression involving boolean connectives (*and*, *or*, etc.) as well as (possibly domain-specific) predicates.

Resource *Resources* (Figure 3.16) are used to represent what would typically be thought of as "objects" in a domain. This includes real-world objects, but can also include any sort of object used in problem solving that does not fall into one of the other categories of abstract PS objects.

In addition to the attributes inherited from *ps-object*, *resources* contain the attribute ACTUAL-OBJECT, which holds a link to the "actual" object as represented in the agent's mental state.

evaluation ← **ps-object**

$$\left[\text{ASSESSMENT} \quad \textit{unstructured}\right]$$

Figure 3.17: Type Description for *evaluation*

situation ← **ps-object**

$$\begin{bmatrix} \text{PS-OBJECTS} & \textit{set(ps-object)} \\ \text{PS-HISTORY} & \textit{list(ps-act)} \\ \text{FOCUS} & \textit{stack(object)} \\ \text{OBJECTIVES} & \textit{objectives-slot} \end{bmatrix}$$

Figure 3.18: Type Description for *situation*

Evaluation Before making decisions in problem solving, an agent typically evaluates each of the options that have been identified. An *evaluation* (Figure 3.17) represents an agent's assessment of a particular PS object within a particular context. The *evaluation* is therefore always associated with a PS object and a context (e.g., which PS object to choose to fill a slot).

A theory of evaluations and their representation is beyond the scope of this thesis, although they present an interesting challenge for future research. We believe, for example, that this is where argumentation could be represented within our model, as identified *evaluations* are wrapped in *fillers* and therefore also are associated with an *evaluations-slot*.

As we are not sure how best to represent the evaluations themselves, we leave the type of the ASSESSMENT attribute unstructured. Analyses we provide that involve *evaluations* will be given as natural language descriptions of the assessment (e.g., *good*).

Situation A *situation* (Figure 3.18), describes the state of a possible world, or more precisely, an agent's beliefs about that possible world. Rather than just packing all state information into a general world-state attribute, we separate out information about problem-solving in the situation, and then have a separate place to store other world beliefs.

The PS-OBJECTS attribute holds a set of all PS objects known to the agent in the situation. This includes domain-specific PS objects (such as objectives, recipes and resources) which the agent can use in problem solving.

The PS-HISTORY attribute records the history of the agent's problem solving. This is a list of problem-solving acts the agent has performed.

The OBJECTIVES attribute encapsulates the problem-solving state of the agent in the situation. This link between *situations* and the PS state is described in Section 3.3.2.

Focus is a known attribute of human communication and is well marked in human communication to make interpretation easier for the hearer [Grosz and Sidner1986; Carberry1990b]. We include this in our single-agent model as well, as we hypothesize that focus is also a feature of single rational agents. This is likely due to resource-boundedness and the need to concentrate resources on a small set of possibilities. Here we model the FOCUS attribute as a stack of general *objects*, following [Grosz and Sidner1986], although, we, as they, admit that a stack is an imperfect representation of focus.[7] Focus can be placed on any type of *object*, including *slots*, *fillers*, or *ps-objects*, depending on whether the focus is finding a value for a slot, evaluating an object, or identifying an object, respectively.

All other agent beliefs about the world are stored in the CONSTRAINTS attribute (inherited from *ps-object*). We originally modeled this as a separate attribute with a set of beliefs, but noticed that this information can be modeled more appropriately with *constraints* and a *constraints-slot*. This turns out to be consistent with our use of

[7] Other proposals for focus structure exist [Lemon, Gruenstein, and Peters2002], but we leave the question of how to best represent focus to future research.

constraints associated with a PS object to identify it. In all but the simplest of domains, an agent will only have partial information about the state of the world and cannot, therefore, model it completely. Instead, an agent can adopt a set of *constraints* on what the current situation is, meaning that it believes that the situation it is trying to model at least conform to the *constraints* that it has adopted towards it. Note, that the use of a *constraints-slot* allows us to model the agent's process of actually deciding which constraints to adopt.

In our model, we only use a single *situation* which describes the current state of the world. However, in future work, we would like to use multiple *situations* to support what-if, possible-world reasoning.

Domain Specialization

The PS model can be specialized to a domain by creating new types that inherit from the abstract PS objects and/or creating instantiations of them. We describe each of these cases separately.

Specialization through Inheritance As described above, inheritance is basically the process of adding new attributes to a previously existing type, and/or specializing the types of preexisting attributes. In our PS model, inheritance is only used for *objectives*, and *resources*. The other abstract PS objects are specialized through instantiation.

Inheriting from *resource* is done to specify domain-specific resource type. As an example, in the MP3 domain, we need to represent songs as resources.

We define a new type *song*, shown in Figure 3.19, which inherits from *resource*. Here we add three new attributes, TITLE, ARTIST and ALBUM so that this information can be recorded in individual *song* instantiations. (Note that the resource types *artist* and *album* also need to be created for the domain, although we do not show them here.) Each of these is a *single-slot*, so that the values of these may be reasoned about in the

$$\textbf{song} \leftarrow \textbf{resource}$$
$$\begin{bmatrix} \text{TITLE} & \textit{single-slot(string)} \\ \text{ARTIST} & \textit{single-slot(artist)} \\ \text{ALBUM} & \textit{single-slot(album)} \end{bmatrix}$$

Figure 3.19: Type Description for *song*

$$\textbf{listen-song} \leftarrow \textbf{objective}$$
$$\begin{bmatrix} \text{SONG} & \textit{single-slot(song)} \end{bmatrix}$$

Figure 3.20: Type Description for *listen-song*

problem-solving process (e.g., in trying to decide who the artist of a particular song is and considering several choices).

Note that it would be also possible to simply use the CONSTRAINTS attribute to store this information for a *song*, however we choose to make these fields explicit, as we want to model them as being in some way an inherent property of the resource.

Inheriting from *objective* is done to specify a particular domain goal, for example, the goal of listing to a song. Typically, additional attributes are *single-slots* of some sort of *resources*, which are used to model parameters of the objective.

We define a new type *listen-song* which represents the goal of listening to a particular song, shown in Figure 3.20. Here we add just one additional attribute SONG, which is a *single-slot* for a *song*. This is a placeholder for deliberation about which song should be listened to.

Specialization through Instantiation All PS object types (including new types created by inheritance) can be further specialized by instantiation, i.e., by assigning values to some set of their attributes. This can be done both at design time (by the domain

modeler) and (as we discuss below) it happens at runtime as part of the problem-solving process itself. For example, we would want to populate our MP3 domain with instantiations of resources: songs, artists, and so forth, as well as recipes: e.g., a recipe for playing a song with an MP3 player. We also instantiate the initial situation with the current state of the world and the agent's beliefs, as outlined in the next section.

3.3.2 The PS State

The PS state models an agent's current problem-solving context. It is represented with a special instance of type *situation* called the *actual-situation*. As the name implies, the *actual-situation* is a model of the agent's beliefs about the current situation and the actual problem-solving context.

The CONSTRAINTS attribute describes the agent's general beliefs about the current state of the world and the PS-OBJECTS attribute contains all of the PS objects which the agent currently knows about — including possible objectives, recipes, resources, etc. The PS-HISTORY attribute is a list of all PS acts the agent has executed, and FOCUS describes the agent's current stack of focus.

The OBJECTIVES attribute contains all of the top-level *objectives* associated with the agent's problem solving process. In the same way as in *recipes*, each of these *objectives* is assigned a status, which we discussed below in more detail. These *objectives* form the roots of individual problem-solving contexts associated with reasoning with, and/or trying to accomplish those *objectives*, and can include all types of other PS objects.

Note that the *actual-situation* contains not only information discovered during problem solving, but also the agent's a priori knowledge.

3.3.3 PS Acts

An agent changes its PS state through the execution of PS acts. There are two broad categories of PS acts: those used in reasoning and those used for commitment. We describe several *families* of PS act types within those categories:

Reasoning Act Families

- *Focus*: Used to focus problem solving on a particular *object*.
- *Defocus*: Removes the focus on a particular *object*.
- *Identify*: Used to identify a *ps-object* as a possible option in a certain context.

Commitment Act Families

- *Adopt*: Commits the agent to an *object* in a certain context.
- *Abandon*: Removes an existing commitment to an *object*.
- *Select*: Moves an *objective* into active execution.
- *Defer*: Removes an *objective* from active execution (but does not remove a commitment to it).
- *Release*: Removes the agent's commitment to an *objective* which it believes it has fulfilled.

Each of these families encompasses a set of actual PS acts. For the remainder of this section, we discuss each of the PS act families and their corresponding acts as well as their effects on the PS state.

Focus/Defocus

As we described above, focus has been shown to be an important part of human communication. We model focus as a stack within the *actual-situation*. The stack can hold pointers to any type of *object*, including *slots*, *fillers*, and *ps-objects*, depending on whether the focus is finding a value for a slot, evaluating an object, or identifying an object, respectively.

Focus is controlled with the following PS acts:

focus(situation-id,object-id)

defocus(situation-id,object-id)

The semantics of these are simple. *focus* pushes the given *object-id* onto the focus stack in the *situation* represented by *situation-id*.[8] *defocus* pops the *object-id* off the stack as well as any *object-ids* above it.

As an example, consider the beginning PS state shown in Figure 3.21 Note that, as PS objects tend to become complex quite quickly, we will often omit attributes that are not relevant to the current discussion. We will also at times, when it is not vital to the discussion, give a description of the contents of an attribute value instead of the formal representation. When this is the case, we will enclose the description in single quotes. Here we have done this for CONSTRAINTS and PS-OBJECTS. Note that within CONSTRAINTS, we use [1] to signify structure sharing.

This initial PS state has the ID *actual-situation* to uniquely identify it as the root of the PS state. It also contains the agent's current beliefs in CONSTRAINTS and knowledge about PS objects in PS-OBJECTS.

[8]Note that we explicitly include a *situation-id* here even though the current CPS model only contains the *actual-situation*. As mentioned above, we hope in future work to extend the model to allow multiple situations for possible-world reasoning.

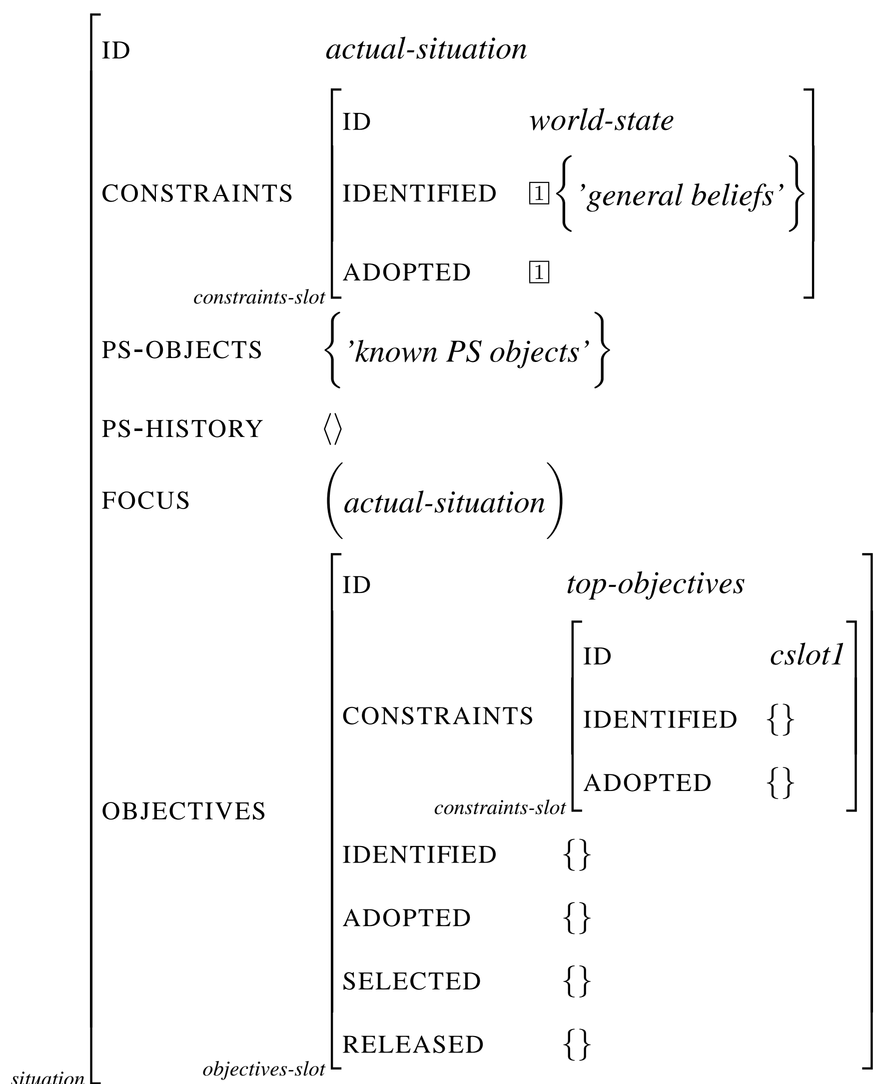


Figure 3.21: A Simple Initial PS State

As no problem solving has yet taken place, PS-HISTORY is an empty set, and FOCUS is simply on the *actual-situation* itself. There are also no *objectives* associated with the actual situation — meaning the agent currently has no objectives and has considered no objectives to pursue.

At this point, our agent decides it needs to set some objectives for itself, and therefore decides to focus on the *objectives-slot top-objectives* by executing the following PS act:

focus(actual-situation,top-objectives)

This has two effects. The first is to put the executed PS act onto the (previously empty) PS-HISTORY list. The second is to push *top-objectives* onto the focus stack.

Identify

PS acts in the *identify* family are used to introduce PS objects into the realm of a problem-solving context. This could either be in identifying previously unknown objects (i.e., objects not listed in PS-OBJECTS within the *situation*), or it could be in identifying a known object as a possible option for filling a certain slot.

All objects must be identified before they can be used further in the PS process. For this reason, PS acts in the *identify* family exist for all PS objects.[9] They are as follows: *identify-objective*, *identify-recipe*, *identify-constraint*, *identify-resource*, and *identify-evaluation*.

The basic syntax of identify acts is

identify-{type}(slot-id,ps-object)

where *type* refers to any of the PS objects in the acts listed above. The *ps-object* parameter is the PS object instance which is being introduced and the *slot-id* parameter gives the *id* of the problem-solving context for which it is being identified. Note that *identify* acts take an actual *ps-object* as an argument, whereas the remaining PS acts take only an *object-id* (pointer to an object). The reasons for this will be discussed below.

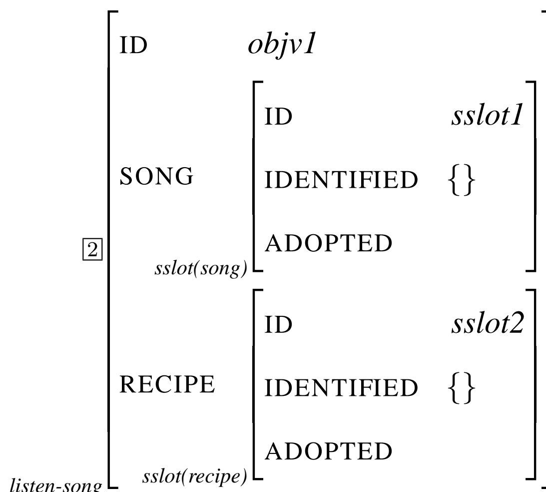
The effect of an *identify* is that the *ps-object* is inserted into the PS-OBJECTS set in the *actual-situation* (if not already there). It is also wrapped in an appropriate *filler* type and inserted into the IDENTIFIED set of the *slot* identified by *slot-id*. If *slot-id* is empty, the *ps-object* is only inserted into the PS-OBJECTS set in the *actual-situation*.

[9] Except *situation* for reasons described above.

Let us assume that at this point, our agent decides it wants to consider an objective of listening to a song and executes

identify-objective(top-objectives, [2] *)*

where [2] abbreviates the following *objective*:[10]



This adds this *listen-song* instance to the PS-OBJECTS attribute of *actual-situation* and add it as well to the IDENTIFIED set of the *objectives-slot*.

These changes, along with those from the focus section are shown in Figure 3.22.

Other *identify* acts have similar effects.

Adopt/Abandon

We treat the PS act families *adopt* and *abandon* together here, as one essentially undoes the other. The syntax of the two is as follows:

adopt-{*type*}*(slot-id,filler-id)*

abandon-{*type*}*(slot-id,filler-id)*

[10]Again here, we have omitted certain features of the *objective* which are not relevant to the current discussion.

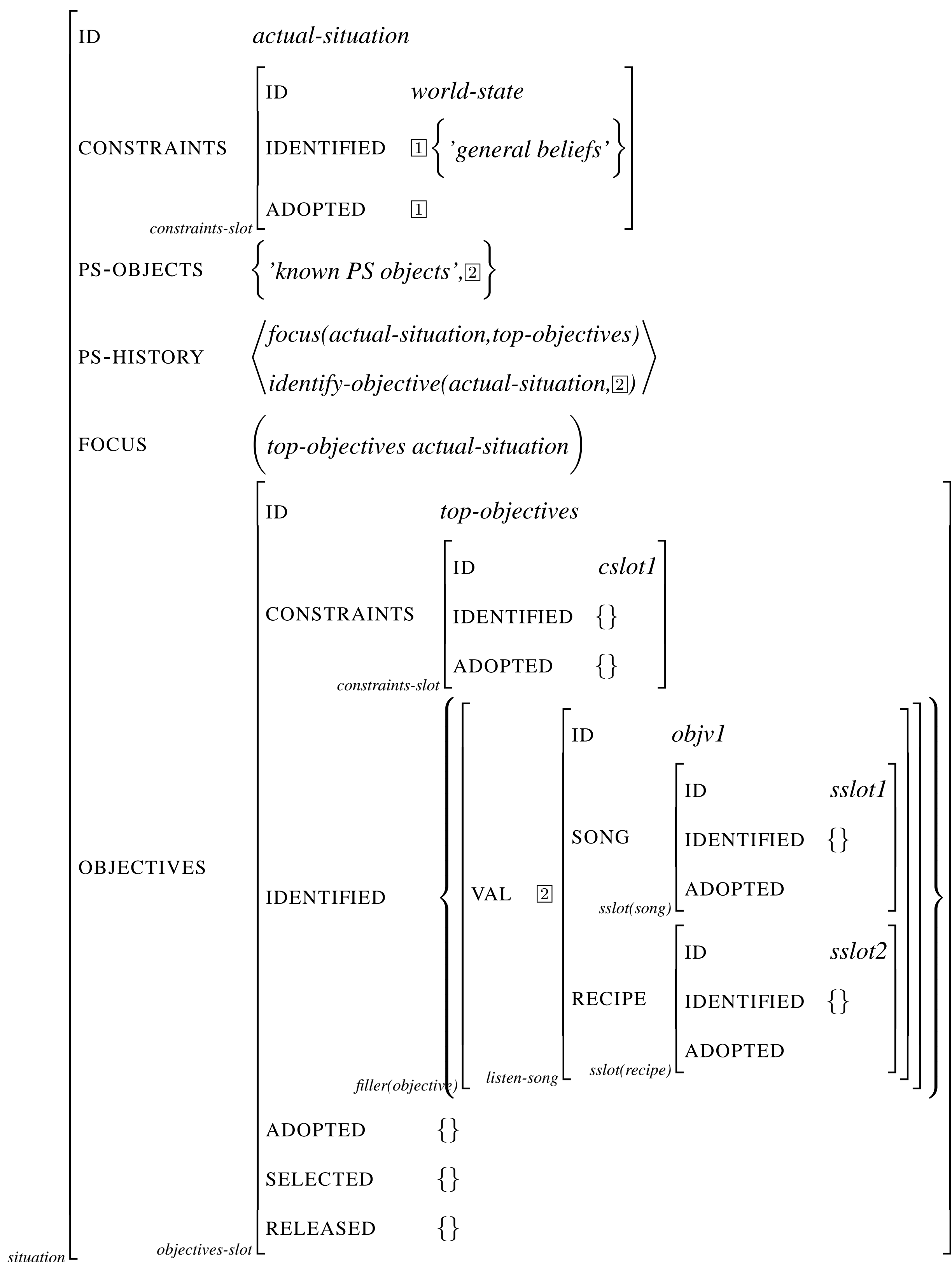


Figure 3.22: The PS State after Executing *identify-objective(actual-situation,2)*

As with *identify*, these two families have types corresponding to most abstract PS objects: *adopt-objective*, *adopt-recipe*, *adopt-resource*, *adopt-constraint*, *adopt-evaluation*, *abandon-objective*, *abandon-recipe*, *abandon-resource*, *abandon-constraint*, and *abandon-evaluation*.

An *adopt* has the effect of adding the *filler* referred to by *filler-id* to the ADOPTED attribute of the *slot* referred to by *slot-id* (either by assigning the value, in the case of a *single-slot* or adding the value to the set of a *multiple-slot*. Note that this requires that the PS object referred to by *filler-id* actually be identified and in the IDENTIFIED set in that context.

An *abandon* basically has the opposite effect. It deletes the object from the ADOPTED attribute. Thus *abandon* requires that the object actually be adopted when the act is executed.

When a PS object is adopted with respect to a slot, it means the agent is committed to that object in that context. For example, for a *recipe*, this means the agent is committed to using that *recipe* for the associated *objective*. The other PS objects are similarly treated.

Adopting *objectives* and Intention Although we do not commit here to any specific theory of commitment or intention (e.g., [Cohen and Levesque1990a; Grosz and Kraus1996]), we need to comment here at least on the meaning of adopted *objectives*.

At the top level (i.e., in the *actual-situation*), adopted *objectives* are those *objectives* which the agent has an intention to achieve.[11] It also follows that if a *recipe* is adopted for one of these *objectives* and if that *recipe* itself has adopted sub-*objectives*, then the agent also intends to achieve those objectives, and so forth. This is not novel.

However, in our model, nothing precludes the agent from adopting *recipes* for *objectives* which it has not adopted at the top level (e.g., those that are only IDENTIFIED.

[11] Again, for this thesis, we try to remain neutral as to exactly what an intention is.

In this case, the agent is committed to *using* an adopted sub-*objective* in that *recipe*, but *not* to actually achieving the sub-*objective*. Of course, if the agent later decides to *adopt* the top-level *objective*, it then automatically intends not only to achieve the top-level *objective*, but also all adopted sub-*objectives*.

This allows us to model what [Carberry, Kazi, and Lambert1992] term *contingent commitments*, where an agent may do planning as part of deciding whether or not to actually decide to achieve the goal itself.

Select/Defer

The act families *select* and *defer* are only used for *objectives*. Their syntax is as follows:

select-objective(slot-id,filler-id)

defer-objective(slot-id,filler-id)

Executing a *select-objective* adds an *objective* to the SELECTED set in the given slot context. *Defer-objective* can then be used to delete an object from the SELECTED set.

Although an agent may have any number of adopted *objectives*, there is only a small subset that is actually being executed at any given point. These are the *objectives* in the SELECTED set. An *objective* does not need to be an atomic action to be selected. Higher-level *objectives* can be marked as selected if the agent believes that it is currently executing some action as part of executing the higher-level *objective*.

Release

The final PS act we discuss here is *release*. As with *select* and *defer*, this is only applicable to *objectives*. The syntax is as follows:

release-objective(slot-id,filler-id)

This act has the effect of moving an *objective* filler from the ADOPTED set to the RELEASED set. Note that the *objective* must first be in the ADOPTED set for this act to be executed.

A rational agent should notice when an *objective* has been successfully achieved and then stop intending to achieve it (cf. [Cohen and Levesque1990a]). The RELEASED set contains those *objectives* which the agent believes have successfully been achieved. Note that this is different than *objectives* which are simply in the IDENTIFIED list, as these were never successfully achieved, at least while the agent had them adopted. Another thing to note is that, in principle, it does not matter if the objective was achieved through the agent's own actions, or by some exogenous event. Fortuitous achievement is achievement nonetheless.

Releasing not only applies to top-level *objectives* but to *objectives* in *recipes* as well. As an agent executes a *recipe* for example, it marks off completed actions by releasing them.

3.4 Collaborative Problem Solving

In the last section we described a problem-solving model for a single agent. In this section, we extend that model to the *collaborative* case, where two agents do problem solving together.

As mentioned above, we have modeled single-agent problem solving at a finer granularity of acts than is typically done. This was to support the granularity at which collaboration occurs between agents (i.e., the contents of utterances). These acts are often more overt in collaborative problem solving since agents must communicate and coordinate their reasoning and commitments in maintaining a *collaborative problem-solving (CPS) state* between them. At the collaborative level, we have CPS acts which operate on PS objects.

We first discuss CPS acts and then the CPS state itself. We then give an example of collaborative problem solving and finish with a discussion of possible compatibilities between our model and the SharedPlans model.

3.4.1 CPS Acts

At the CPS level are CPS acts which apply to the PS objects, paralleling the single-agent PS model. In order to distinguish acts at the two levels, we append a *c-* before CPS acts, creating *c-adopt-resource*, *c-select-objective*, *c-identify-constraint*, and so forth. CPS acts have the similar syntax and effects as those at the PS level and we will not redefine them here.[12] Later we do discuss slight changes to the *situation* object and the semantics of the *identify* act.

As we move to the collaborative level, however, we encounter an important difference. Whereas at the PS level, an agent could change its own PS state, at the collaborative level, an agent cannot single-handedly make changes to the CPS state. Doing so requires the cooperation and coordination of both agents. This means that no agent can directly execute a CPS act. Instead, CPS acts are generated by the individual *interaction acts* (IntActs) of each agent. An IntAct is a single-agent action which takes a CPS act as an argument. These are used by the agents to negotiate changes to the CPS state.

The IntActs are *begin*, *continue*, *complete* and *reject*.[13] These are defined by their effects and are similar in spirit to the actions in the model of grounding proposed in [Traum1994].

[12] Several formal models of intention [Levesque, Cohen, and Nunes1990; Grosz and Kraus1996] have explored intentional differences between single-agent plans and group plans. This is beyond the scope of this thesis.

[13] In earlier papers [Allen, Blaylock, and Ferguson2002; Blaylock, Allen, and Ferguson2003], we used *initiate* instead of *begin*. However, we had to change this because of a naming clash with part of the dialogue model in Chapter 4.

An agent beginning a new CPS act proposal performs a *begin*. For successful generation of the CPS act, the proposal is possibly passed back and forth between the agents, being revised with *continues*, until both agents finally agree on it, which is signified by an agent *not* adding any new information to the proposal but simply accepting it with a *complete*. This generates the proposed CPS act resulting in a change to the CPS state.

At any point in this exchange, either agent can perform a *reject*, which causes the proposed CPS act — and thus the proposed change to the CPS state — to fail. This ability of either agent to negotiate and/or reject proposals allows our model to represent not just the master-slave collaboration paradigm, but the whole range of collaboration paradigms (including mixed-initiative).[14]

Sidner's negotiation language [Sidner1994; Sidner1994] has similar goals to our interaction acts. However, as pointed out in [Larsson2002], the language conflates proposal acceptance (similar to our interaction level) and communicative grounding, i.e., coordination of the reliability of a communicative signal (which the CPS model assumes to be handled by lower-level communicative behavior which is outside the model). In addition, several actions in Sidner's language (such as *ProposeAct*, which proposes that the other agent perform an action) include aspects of acts that we model as CPS acts.

3.4.2 CPS State

In the single-agent case, we modeled the PS state as part of the agent's mental state. This is not possible in the collaborative case. Instead, we model the CPS state as an emergent property of the mental states of the collaborating agents. In this way, the CPS state can be seen as part of the agents' *common ground* [Clark1996]. Thus each

[14] Again it is important to note that our purpose is not to specify problem-solving behavior for a particular agent, but rather to provide a model which allows collaboration between agents with (possibly) very different behavior. In the master-slave paradigm, the slave agent will simply not use the *reject* act. The fact that rejections do not occur in that interaction is unimportant to the CPS model.

c-situation ← ps-object

PENDING-PS-OBJECTS	*set(ps-object)*
PS-OBJECTS	*set(ps-object)*
PS-HISTORY	*list(interaction-act)*
FOCUS	*stack(object)*
OBJECTIVES	*objectives-slot*

Figure 3.23: Type Description for *c-situation*

agent has a mental model of what they believe the CPS state to be. With this model, it is possible that agents' CPS states get "out of sync" because of misunderstanding or miscommunication. We leave the issue of how such problems are resolved to future research, although the reader is referred to [Clark1996] for a good discussion of possible solutions based on human communication. In this discussion, we assume that communicative signals are always received and properly understood — a restriction we remove in the next chapter.

In the CPS model, we are able to reuse all of the PS objects as they are, except one. In order to accommodate a collaborative version of the PS state, we introduce a collaborative situation *c-situation* to replace *situation*. The definition is shown in Figure 3.23.

This differs from *situation* in two ways. First, we have changed the type of PS-HISTORY to be a list of *interaction-acts* instead of *ps-acts*, as this is the new atomic level of communication. Now, each time an IntAct is executed, it is automatically added to the PS-HISTORY list.

Second, we have introduced a new feature PENDING-PS-OBJECTS. This is used to store *ps-objects* which are objects under negotiation via a *c-identify* act. This is necessary since *ps-objects* are not officially added to the PS-OBJECTS attribute of the C-

SITUATION until the *c-identify* act has been successfully generated. Thus PS-OBJECTS under negotiation can be accessed and changed without being officially identified.

All newly mentioned *ps-objects* are added to the PENDING-PS-OBJECTS set when an IntAct introduces them. When a *continue* IntAct makes a change to a pending *ps-object*, it is changed in this set.

Finally, we slightly change the semantics of the *c-identify* acts. When a *c-identify* act is successfully generated, we move all of the new *ps-objects* from the PENDING-PS-OBJECTS set to the PS-OBJECTS set (before we added them to PS-OBJECTS directly).

With these slight changes, we can now represent the collaborative problem-solving state.

3.4.3 An Example

We show here one short example of the CPS model in an agent exchange. We then show more examples in the next chapter, where we tie the model to natural language dialogue. Examples are much easier to follow when shown in natural language dialogue.

As displayed feature structures tend to become large fairly quickly, we will use a set of abbreviations for type and feature names for this example as well as in examples in Chapter 4. The abbreviations are shown in Table 3.1.

In this example, we will stay in the MP3 player domain mentioned above. Here two agents (A and B) are collaborating on use of an MP3 player.

At the beginning of the exchange, Agent A decides it wants to listen to a song (although it does not yet know which). It executes the following interaction acts:[15]

$$\textit{begin}_1(\textit{c-identify-objective}(\text{STATE} \mid \text{OBJVS} \mid \text{ID}, \boxed{2}\,))$$
$$\textit{begin}_2(\textit{c-adopt-objective}(\text{STATE} \mid \text{OBJVS} \mid \text{ID}, \boxed{2} \mid \text{ID}))$$

[15]Note, for reasons of clarity, we ignore issues of focus in this example. We revisit focus in the collaborative setting in the next chapter.

types		features	
full	**abbr.**	**full**	**abbr.**
c-situation	csit	action-constraints	acons
constraint	con	actions	acts
constraints-slot	cslot	actual-object	aobj
evaluation	eval	adopted	aptd
evaluations-slot	eslot	constraints	cons
filler	fill	evaluations	evals
object	obj	expression	exp
ps-object	psobj	identified	ided
single-slot	ss	objectives	objvs
objective	objv	pending-ps-objects	pend
objectives-slot	oslot	ps-history	pshist
recipe	rec	ps-objects	psobjs
resource	res	recipe	rec
situation	sit	released	reld
		selected	seld
		value	val

Table 3.1: Abbreviations for Type and Feature Names

Note that here, instead of using the ids (*top-objectives* and *objv1*) for the context parameters of the acts, we use absolute paths (STATE | OBJVS | ID and [2] | ID). We introduce a constant STATE, which always identifies the root of the CPS state. Here, [2] abbreviates an empty *listen-song* objective, shown in Figure 3.24. We also subindex the IntActs here to allow us to easily match them up with later IntActs operating on the same CPS acts (see below).

The CPS state resulting from the two IntActs is shown in Figure 3.25. In order to show feature structures more compactly, we will often omit those features which are empty or unimportant for discussion purposes. An abbreviated version of the CPS State from Figure 3.25 is shown in Figure 3.26.

There are several things to note here. First, we use a path to refer to the APTD feature within CONS. In the full version, this set is actually structure-shared with CONS | IDED. However, as we note earlier, for an object to be adopted, it must be identified. Thus this

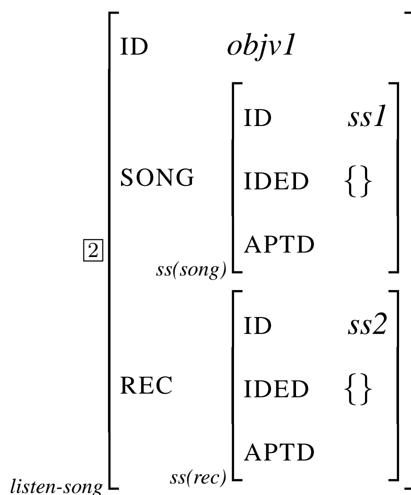


Figure 3.24: *listen-song* Objective 2

.

is a common phenomenon that we omit as redundant information.

Also note that we still display the OBJVS feature, even though it is empty. This is of course because we will need it soon in our example.

Now, back to our example. Notice that only the PEND and PSHIST attributes are set at this point. Although other parts of the CPS can only be changed through negotiation, a successful communication (execution of an IntAct) *does* change the state to represent that that communication has occurred — without any negotiation (cf. [Clark1996]).

Agent B now decides to accept these proposals and executes the following interaction acts:[16]

$$complete_1$$
$$complete_2$$

which complete and therefore generate the corresponding CPS acts:

[16]We will usually omit the act arguments and use coindexing instead.

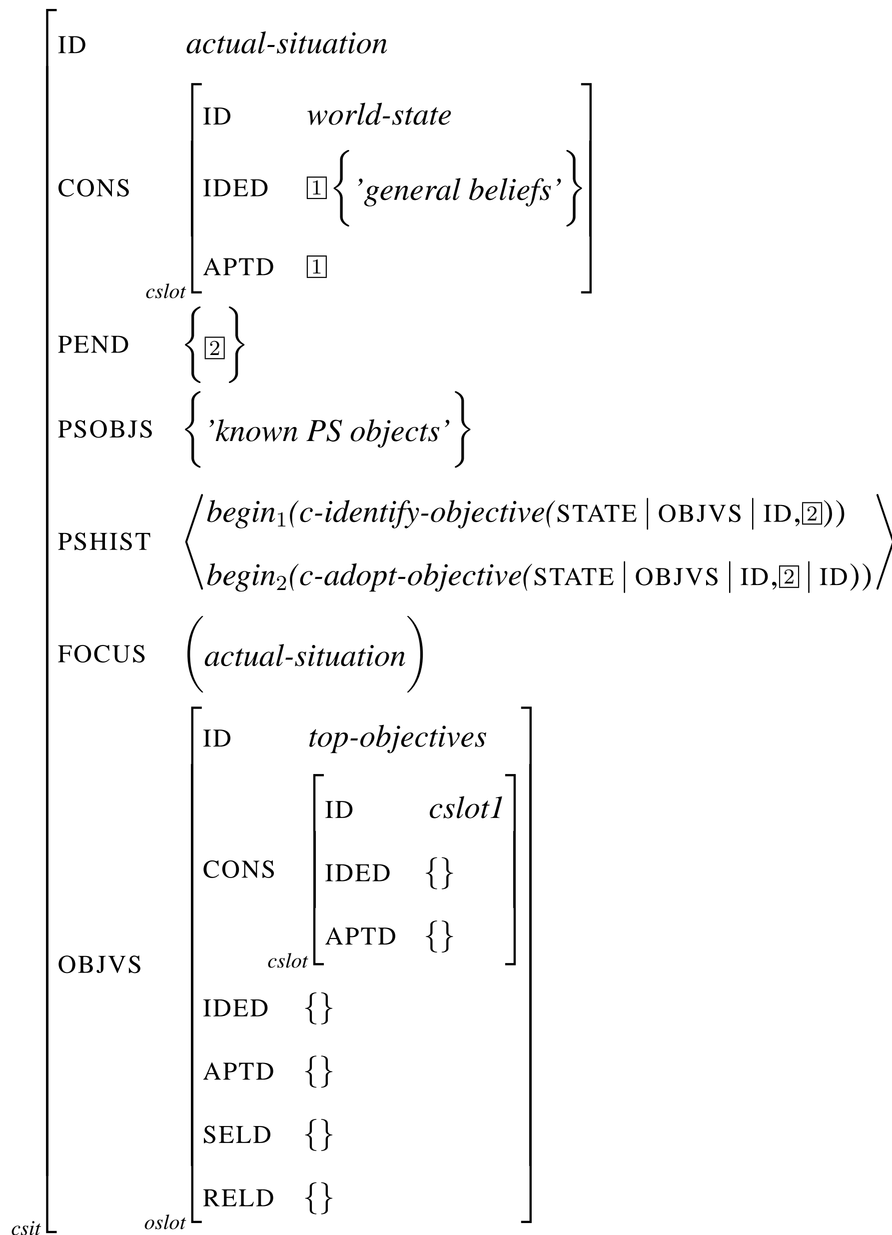


Figure 3.25: The CPS State after A's First Turn

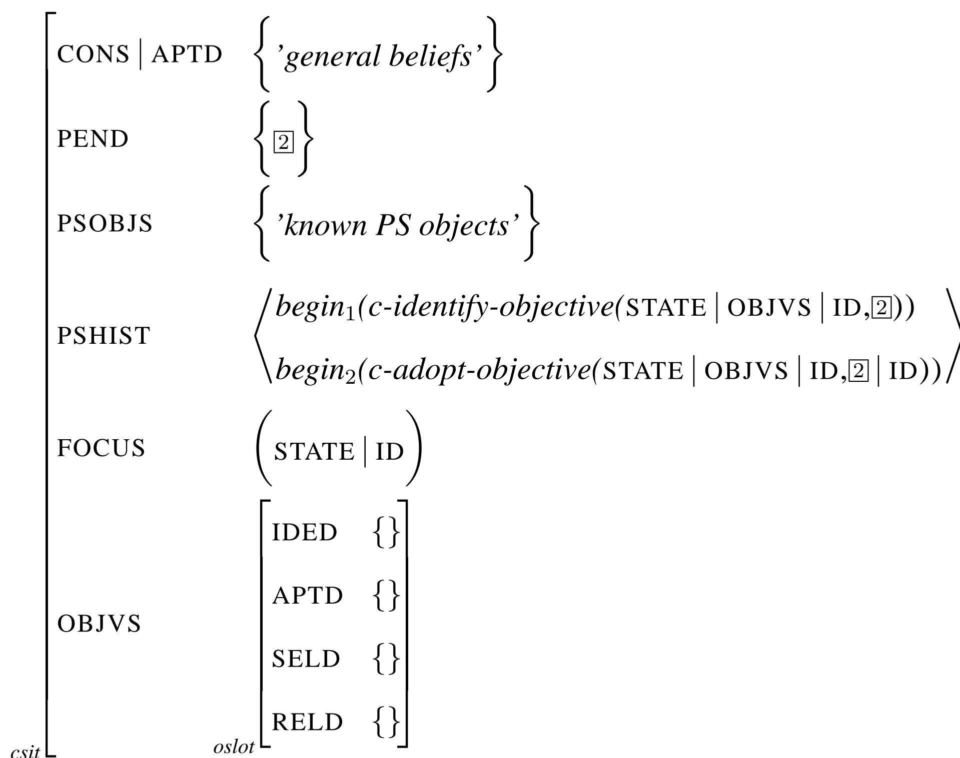


Figure 3.26: The Abbreviated Version of Figure 3.25

*c-identify-objective(*STATE | OBJVS | ID, 2 *)*

*c-adopt-objective(*STATE | OBJVS | ID, 2 | ID*)*

This results in the CPS state shown in Figure 3.27.

Note that here the items in PEND have been moved to PSOBJS as they have now been identified. Also note that 2 has been wrapped in a *filler* (3) and is both in the IDED and APTD sets of OBJVS as a result of being both identified and adopted.

At this point, B has a suggestion for a song — "Yesterday" by the Beatles:

$begin_3$*(c-identify-resource(* 2 | ID, 4 *))*

$begin_4$*(c-adopt-resource(* 2 | ID, 4 | ID*))*

The structure corresponding to 4 is shown in Figure 3.28.

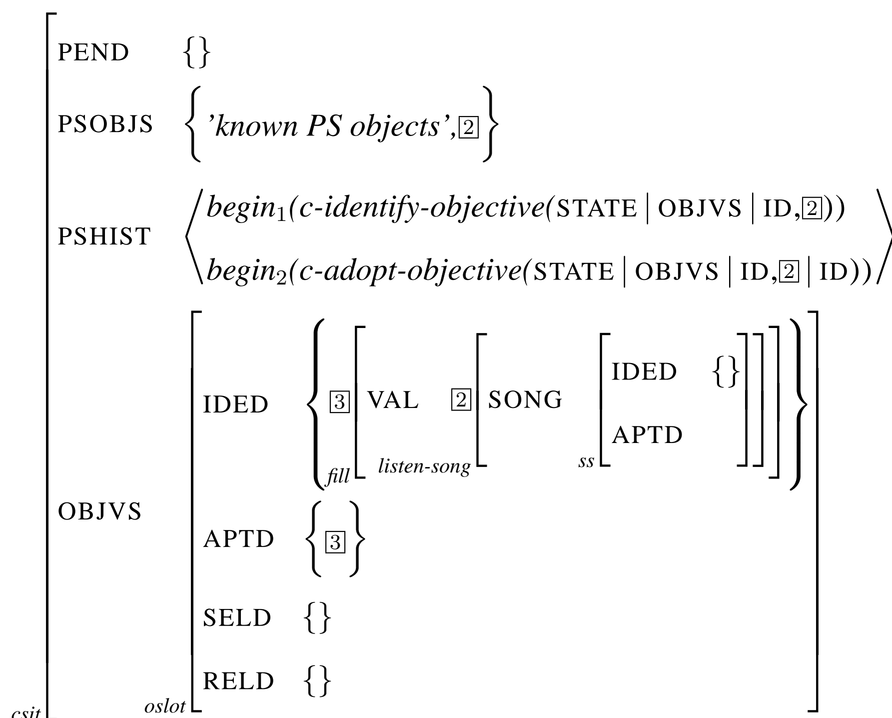


Figure 3.27: The CPS State after B's First Turn

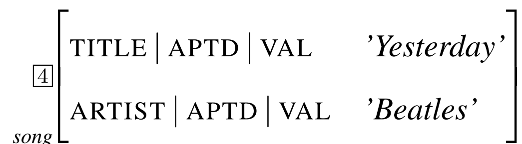


Figure 3.28: *song* Resource 4 — "Yesterday" by the Beatles

Note that 4 already has several of its slots filled (*title* and *artist*). As a short-cut, human communication often uses *compound objects* which already have several decisions premade, in this case the identification and adoption of the song TITLE and ARTIST. Also note that we have omitted the definition of a real *artist* type here and just gloss the artist name with a string.

At this point, Agent A decides it likes the idea and completes the CPS acts:

$$complete_3$$

$$complete_4$$

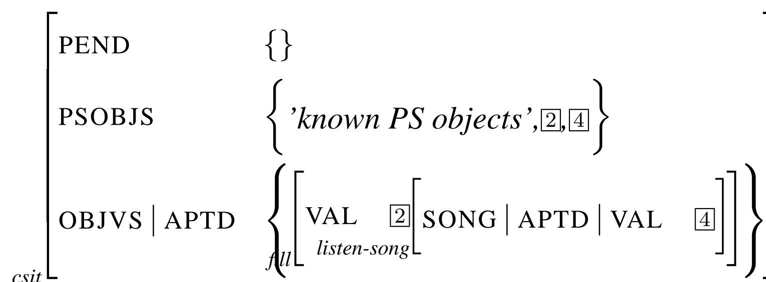


Figure 3.29: The CPS State after A's Second Turn

producing the CPS state shown in Figure 3.29.

Note that the changes here were made within the *listen-song* objective itself (2), where the *song* was wrapped in a *filler*, and put in the IDED set and was assigned as the APTD value.

3.4.4 Possible Compatibility with SharedPlans

As we mentioned in Chapter 2, we believe that at a high level, our model is compatible with the SharedPlan formalism [Grosz and Kraus1996]. In fact, one way of looking at our model is an elaboration of SharedPlan operators. Adoption, evaluation, etc. of *objectives* actually resides at a level higher than the SharedPlan model, since SharedPlans assumes that a high-level goal has already been chosen. The adoption, evaluation, etc. of recipes can be seen as a further elaboration of the Select_Rec_GR operator. Most other acts, such as adopting resources, evaluating actions, etc. provide the details of the Elaborate_Group operator. We leave it as a topic of future research to see if the approaches can indeed be unified.

3.5 Conclusions and Future Work

In this chapter, we have presented a model of collaborative problem solving for agents which (1) covers collaboration about a wide range of agent behavior and (2) allows col-

laboration using a wide variety of collaborative paradigms, including mixed-initiative where agents may freely negotiate their decisions.

The model is novel in that it models agent problem solving at the granularity of communication and can be used for general heterogeneous agent-agent communication/collaboration. Agents can be heterogeneous in the sense that they implement different behavioral cores, as long as they are able to collaborate within the bounds of the model. As we show in the next chapter, the model is also general enough to support human-agent collaboration.

The model is also able to represent a number of problem solving phenomena at the granularity of the decision-making *process* and not just the decisions made. We model the identifications of possible alternatives for a given role, as well as the possible constraining of values to consider for the role. We also model the contextual evaluation of values for a given role. In addition, since we model the identification of objects without attaching commitment to them, we model contingent planning, where agents may plan before actually intending to execute the plan.

Important future work includes formalization of the model at a level of single-agent beliefs, intentions and desires, possibly within the SharedPlans model. We would also like to expand the CPS process to include the step of *team formation* [Wooldridge and Jennings1999], when agents decide to collaborate in the first place. This would allow the model to cover such things as requests for help.

Another item that seems to be lacking in the model is a representation of which agent (or group of agents) is assigned to plan/execute which objectives. Right now, such decisions for execution could be modeled with an ACTOR attribute on each objective, although it may make sense to treat it somehow specially.

Finally, another interesting direction is exploring the use of the single-agent PS model in (keyhole) agent modeling.

4 Modeling Dialogue as Collaborative Problem Solving

In this chapter,[1] we present a dialogue model based on the collaborative problem-solving model presented in the last chapter. The model, as presented in Chapter 3, can be used as a language for communication between artificial agents, with interaction acts serving as "utterances". There are several things that must be done, however, to make the CPS model usable for modeling *human* communication.

First, we must provide a link from interaction acts to natural language utterances. Second, the CPS model makes the assumption that interaction acts are always successfully received and understood by the other agent. This, of course, is not the case for human dialogue,[2] where mishearing and misunderstanding is more of a rule than an exception. In this chapter, we provide these necessary parts to turn the CPS model into a dialogue model.

The rest of this chapter is as follows: in Section 4.1, we deal with the first problem by forming a link between the CPS model and natural language utterances through a definition of *communicative intentions*. In Section 4.2, we deal with the second problem by tying the model together with a theory of *grounding*. In order to show the coverage of the model, we show several examples of the model applied to dialogues in Section 4.3.

[1] Some contents of this chapter were reported in [Blaylock and Allen2005a].

[2] It also really is not realistically the case for artificial agent communication either.

In Section 4.4 we conclude and mention future work.

4.1 Collaborative Problem Solving and Communicative Intentions

Communicative intentions describe what a speaker wants a hearer to *understand* from an utterance [Grice1969]. In this sense, communicative intentions can be very different from a speaker's actual intentions, or what a speaker wants to *accomplish* by making an utterance. This can include things such as manipulation and deception, which are not intended to be perceived by the hearer.

When a speaker has decided on his communicative intentions, he must then encode them in language (e.g., words and sounds) with which they are transfered to the hearer, who then must decode them and (hopefully) recover the original communicative intentions associated with the utterance. No matter what their actual intentions are, speakers will alway try to encode their communicative intentions in such a way that they are easily decodable by hearers. This is why communication works, even when one party wants to deceive the other.

In our agent-based dialogue mode, we represent communicative intentions with interaction acts.[3] In other words, each utterance is associated with a set of IntActs. In this way, we are basically modeling dialogue as negotiation about changes to a collaborative problem-solving state. Each utterance is a move in this negotiation, as described with IntActs in the last chapter.

As an example, consider the following utterance with its corresponding interpretation (in a typical context)

---

[3]This definition will be expanded in the next section.

1 A: Let's listen to a song.
begin$_1$*(c-identify-objective(*STATE | OBJVS | ID, $\boxed{1}$ *[blank listen-song]))*
begin$_2$*(c-adopt-objective(*STATE | OBJVS | ID, $\boxed{1}$ | ID*))*
begin$_3$*(c-focus(*STATE | ID, $\boxed{1}$ | ID*))*

where $\boxed{1}$ abbreviates an empty *listen-song* objective like the one shown in Figure 3.24.

By assigning these IntActs, we claim that, in the right context, utterance (1) has three communicative intentions (corresponding to the three IntActs):

1. To propose that listening to a song be considered as a possible top-level objective.
2. To propose that this objective be adopted as a top-level objective.
3. To propose that problem-solving activity be focused on the listen-song objective (e.g., in order to specify a song, to find a recipe to accomplish it, . . .).

That these are present can be demonstrated by showing possible responses to (1) which reject some or all of the proposed CPS acts. Consider the following possible responses to (1), tagged with corresponding communicative intentions:

2.1 B: OK.
complete$_1$
complete$_2$
complete$_3$

This is a prototypical response, which completes all three acts.

2.2 B: No.
complete$_1$
reject$_2$
reject$_3$

This utterance rejects the last two CPS acts (*c-adopt-objective* and *c-focus*), but actually completes the first CPS act (*c-identify-objective*). This means that B is actually accepting the fact that this is a *possible* objective, even though B rejects *committing* to

it. The next possible response shows this contrast:

2.3 B: I don't listen to songs with clients.
reject$_1$
reject$_2$
reject$_3$

Here all three CPS acts are rejected. The *c-identify-objective* is rejected by claiming that the proposed class of objectives is situationally impossible, or inappropriate. Note that it is usually quite hard to reject acts from the *c-identify* family.[4]

2.4 B: OK, but let's talk about where to eat first.
complete$_1$
complete$_2$
reject$_3$

This example helps show the existence of the *identify(c-focus)* act. Here B completes the first two CPS acts, accepting the objective as a possibility and also committing itself to it. However, here the focus move is rejected, and a different focus is proposed (IntAct not shown).

This method of finding responses to reject certain CPS acts proves to be a useful way of helping annotate utterances with their communicative intentions, and we have used it in annotating the examples shown in this chapter.

4.2 Grounding

The account of communicative intentions given in the last section is not quite correct. It makes the simplifying assumption that utterances are always correctly heard by the

[4]In our current CPS model, when a *c-identify* is rejected, the corresponding object then only exists in the PS-HISTORY attribute of the *actual-situation*. It is likely that it would be beneficial to record this somewhere within the corresponding slot as well (e.g., to make sure the same suggestion isn't made twice). We leave this question to future research.

hearer and that he also correctly interprets them (i.e., properly recovers the communicative intentions). In human communication, mishearing and misunderstanding can be the rule, rather than the exception. Because of this, both speaker and hearer need to *collaboratively* determine the meaning of an utterance (i.e., the communicative intentions). This occurs through a process called *grounding* [Clark1996].

In this section, we expand our definition of communicative intentions to handle grounding. To do this, we merge our CPS model with a theory of utterance meaning based on Clark's work called *Conversation Acts Theory*. We first introduce Conversation Acts Theory and then use it to expand our definition of communicative intentions.

4.2.1 Conversation Acts Theory

Traum and Hinkelman [1992] proposed Conversation Acts as an extension to speech act theory. Similar to our provisional model of communicative acts, theories based on speech acts typically made the assumption that utterances are always heard and understood. To overcome this, Traum and Hinkelman defined Conversation Acts on several levels, which describe different levels of the communicative process. Table 4.1 shows the different levels and acts, which we briefly describe here and then show an example using Conversation Acts.

Core Speech Acts

A major contribution of Conversation Acts is that it takes traditional speech acts and changes them into Core Speech Acts, which are multiagent actions requiring efforts from the speaker and hearer to succeed. To do this, they define a *Discourse Unit* (DU) to be the utterances which contribute to the grounding of a Core Speech Act. These are similar to Clark's *contributions* [Clark1996].

Discourse Level	Act Type	Sample Acts
Sub UU	**Turn-taking**	take-turn keep-turn release-turn assign-turn
UU	**Grounding**	Initiate Continue Ack Repair ReqRepair ReqAck Cancel
DU	**Core Speech Acts**	Inform WHQ YNQ Accept Request Reject Suggest Eval ReqPerm Offer Promise
Multiple DUs	**Argumentation**	Elaborate Summarize Clarify Q&A Convince Find-Plan

Table 4.1: Conversation Act Types [Traum and Hinkelman1992]

Argumentation Acts

Conversation Acts also provides a place for higher-level *Argumentation Acts* which span multiple DUs. As far as we are aware, this level was never well defined. Traum and Hinkelman give examples of possible Argumentation Acts, including such things as rhetorical relations (e.g., [Mann and Thompson1987] and plan construction plans (e.g., [Litman and Allen1990]).

Grounding Acts

The most important part of Conversation Acts for our purposes here are *Grounding Acts* (GAs). These are single-agent actions at the *Utterance Unit* (UU) level, used for the grounding process. The GAs are as follows:

Initiate The initial part of a DU.

Continue Used when the initiating agent has a turn of several utterances. An utterance which further expands the meaning of the DU.

Acknowledge Signals understanding of the DU (although not necessarily *agreement*, which is at the Core Speech Act level).

Repair Changes some part of the DU.

ReqRepair A request that the other agent repair the DU.

ReqAck An explicit request for an acknowledgment by the other agent.

Cancel Declares the DU as 'dead' and ungrounded.

These form part of Traum's computational theory of grounding [Traum1994], which uses finite state automata to track the state of grounding for DUs in a dialogue.

Turn-taking Acts

At the lowest level are *Turn-taking Acts*. These are concerned with the coordination of speaking turns in a dialogue. A UU can possibly be composed of several Turn-taking Acts (for example, to take the turn at the start, hold the turn while speaking, and then release it when finished).

Example

As an example, Traum and Hinkelman annotate part of a dialogue from the TRAINS-91 corpus [Gross, Allen, and Traum1992] to illustrate Conversation Acts. In Figures 4.1 and 4.2, we show a section of their example of GA and Core Speech Act annotations.[5]

[5] The original used S for the system and M for 'manager'. We have changed this to the more typical U for 'user'.

GA$_{DU\#}$	UU#	: Utterance
init$_1$	1.1	U: okay, the problem is we better ship a boxcar of oranges to Bath by 8 AM.
ack$_1$	2.1	S: okay.
init$_2$	3.1	U: now ... umm ... so we need to get a boxcar to Corning, where
		: there are oranges.
init$_3$	3.2	: there are oranges at Corning
reqack$_3$	3.3	: right?
ack$_3$ init$_4$	4.1	S: right.
ack$_4$ init$_5$	5.1	M: so we need an engine to move the boxcar
reqack$_5$	5.2	U: right?
ack$_5$ init$_6$	6.1	S: right.

Figure 4.1: Example of Conversation Acts: Grounding Acts [Traum and Hinkelman1992]

DU#	Core Speech Act types	Included UUs
1	inform^M suggest(goal)^M accept^S	1.1 1.2
2	inform^M suggest^M	3.1
3	check^M ?suggest^M	3.2 3.3 4.1
4	inform-if^S ?accept^S	4.1 5.1
5	check^M	5.1 5.2 6.1

Figure 4.2: Example of Conversation Acts: Core Speech Acts [Traum and Hinkelman1992]

Figure 4.1 shows the GAs associated with each UU (subscripted with the number of the DU they contribute to). Figure 4.2 shows the Core Speech Acts performed in each DU (superscripted with the initiating party).

Discussion

One main contribution of Conversation Acts Theory is that it models dialogue with utterances making simultaneous contributions at several different levels. While we believe that dialogue should be modeled as several levels, we see several difficulties with the theory as it now stands.

First, the theory only specifies act *types* at the various levels, but not their content. This is true even at the interface between levels. For example, GAs are modeled as

negotiation about the meaning of a DU, but it is unclear exactly which part the meaning of DU 1 (if the $init_1$ in the example is initializing (*inform*, *suggest*, *accept*)).

Also, although this model improves on speech act theory by modeling speech acts as multiagent actions, it still suffers from some of the difficulty of speech acts. In particular, Conversation Acts Theory does not attempt to define a (closed) set of allowable Core Speech Acts. It is in fact unclear if such a closed set of domain-independent speech acts exist (cf. [Clark1996; Di Eugenio et al.1997]). In practice, this fact has lead to many different proposed taxonomies of speech or dialogue acts, many of which are domain dependent (e.g., [Allen and Core1997; Alexandersson et al.1998] — also cf. [Traum2000]).

Finally, as mentioned above, the Argumentation Acts level was never well defined. In [Traum and Hinkelman1992], Argumentation Acts are described vaguely at a level higher than Core Speech Acts which can be anything from rhetorical relations to operations to change a joint plan.

4.2.2 Defining the Dialogue Model

In constructing our agent-based dialogue model, we first take Conversation Acts as a base. In particular, we model communicative intentions at several simultaneous levels, which more or less correspond to those used in Conversation Acts. In addition to this, we expand and concretize several of the levels using the collaborative problem-solving model discussed in the previous chapter and above. This allows us to overcome several of the difficulties of Conversation Acts mentioned above.

The levels we model are: Turn-taking Acts, Grounding Acts, Interaction Act, and CPS Acts. We discuss each in turn and then reinterpret the previous example with our model.

Turn-taking Acts

At the sub-utterance level, we use the turn-taking model as it is in Conversation Acts. We will not refer to it further in our examples, as it is not the focus of this thesis.

Grounding Acts

At the utterance unit level (UU), we use the Grounding Act *types* as they are defined in Conversation Acts. We extend this and define *contents* for these acts, namely an Interaction Act.[6]

Interaction Acts

At the discourse unit level (DU), we depart from Conversation Acts. Instead of using Core Speech Acts, we use IntActs, as described in Section 4.1. Unlike Core Speech Acts, these are not just labels, but also contain content (instantiated CPS acts).

CPS Acts

Finally, we propose the use of CPS acts at the level of Argumentation Acts. These are a natural fit, as a number of IntActs need to be executed by different agents in order to generate a CPS act, which then makes changes to the CPS state. This gives us a natural segmentation of discourse units.

TRAINS Example Revisited

To show concretely, how these (last three) levels fit together, we revisit Traum and Hinkelman's example from above, interpreting it in the agent-based model. Figure 4.3

[6] A Grounding Act could also theoretically take another Grounding Act as an argument, as in meta-repairs and so forth [Traum1994]. For simplicity, we have decided to avoid these cases at this stage. We plan to add support for meta-grounding in the future.

1.1 U: okay, the problem is we better ship a boxcar of oranges to Bath by 8 AM.

$init_1(begin_1(c\text{-}identify\text{-}objective(\text{STATE} \mid \text{OBJVS} \mid \text{ID}, \boxed{1})))$

$init_2(begin_2(c\text{-}adopt\text{-}objective(\text{STATE} \mid \text{OBJVS} \mid \text{ID}, \boxed{1} \mid \text{ID})))$

$init_3(begin_3(c\text{-}focus(\text{STATE} \mid \text{ID}, \boxed{2} \mid \text{ID})))$

2.1 S: okay.

ack_1

ack_2

ack_3

$init_4(complete_1)$

$init_5(complete_2)$

$init_6(complete_3)$

3.1 U: now ... umm ... so we need to get a boxcar to Corning, where there are oranges.

ack_4

ack_5

ack_6

$init_7(begin_4(c\text{-}identify\text{-}recipe(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2})))$

$init_8(begin_5(c\text{-}adopt\text{-}recipe(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2} \mid \text{ID})))$

$init_9(begin_6(c\text{-}focus(\text{STATE} \mid \text{ID}, \boxed{2} \mid \text{ID})))$

3.2 U: there are oranges at Corning

$init_{10}(begin_7(c\text{-}identify\text{-}constraint(\text{STATE} \mid \text{CONS} \mid \text{ID}, \boxed{3})))$

$init_{11}(begin_8(c\text{-}adopt\text{-}constraint(\text{STATE} \mid \text{CONS} \mid \text{ID}, \boxed{3} \mid \text{ID})))$

3.3 U: right?

$reqack_{10}$

$reqack_{11}$

4.1 S: right.

ack_{10}

ack_{11}

$init_{12}(complete_7)$

$init_{13}(complete_8)$

Figure 4.3: The TRAINS Example Interpreted with the Agent-based Model

shows the dialogue marked up with instantiated grounding acts. We first discuss the dialogue at the grounding level, and then at the problem-solving level.

Grounding Our analysis at the grounding level is basically unchanged from that of Traum and Hinkelman as shown in Figure 4.1. We therefore only briefly describe it. The main difference between the two accounts is that we associate a GA with each

individual IntAct, and therefore have more instances in several cases.

In UU 1.1, the user initiates three IntActs, which the system acknowledges in UU 2.1. Note that, only at this point are the effects of the IntActs (as described in Chapter 3) valid. This means that, only after UU 2.1 are, for example, IntActs 1, 2 and 3 placed in the PS-HISTORY list of the CPS state.

UU 2.1 also initiates the corresponding *complete* IntActs to those initiated in UU 1.1; these are acknowledged in UU 3.1.

UU 3.1 also inits three IntActs (7, 8, and 9), which are never grounded,[7] and thus do not result in any successfully executed IntActs, and thus no changes to the CPS state.

In UU 3.2, the user inits new IntActs for which he explicitly requests an acknowledgment in UU 3.3. Finally, UU 4.1 inits two completes, which are acknowledged by the subsequent utterance by the user (not shown). Note that, again, these completes are not valid until grounded. Thus the CPS state after UU 4.1 will reflect the state as if those completes did not yet occur.

Problem Solving At the problem-solving level, the user proposes the adoption (and identification) of an objective of shipping oranges in UU 1.1. (Again, these do not become active until grounded in UU 2.1). He also proposes that problem-solving focus be placed on that objective (i.e., in order to work on accomplishing it). The proposed objective is shown in Figure 4.4, and deserves some explanation.

The type of the objective is *ship-by-train*, which we have just invented for this example.[8] It introduces two new attributes to the *objective* class: an item to be shipped and a destination. As the abbreviated form of the objective shows, there are three main components to the objective as it has been introduced by the user. First of all, it has a pre-adopted destination — Bath (modeled as a location with a NAME). Second, the

[7] Although see discussion in [Traum and Hinkelman1992].

[8] In all examples, we invent simple-minded domain-specific object types as we need them.

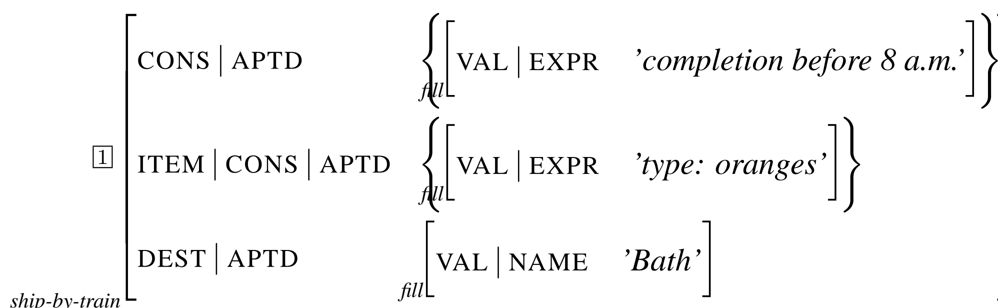


Figure 4.4: Contents of *objective* 1

item to be shipped has not yet been determined, but a constraint has been put on possible values for that slot — they must be of type *oranges*.

Recall from the previous chapter that this was one of the motivations for introducing slots in the model. Notice here that the constraint is not put on a particular instance of oranges, rather it is put on the *single-slot* itself. Constraints on a slot are adopted to restrict the values considered (e.g., identified) as possible fillers.

Finally, a constraint has also been placed on the objective itself — that it be completed by 8 a.m. Note the difference here between placing a constraint on a *ps-object* versus placing it on a *slot*, as just discussed. As mentioned in the previous chapter, objectives and resources are usually extended by the addition of additional resources (as we did here with the item and destination). However, it is also possible to further define an objective or resource by placing a constraint on it. This seems to work well in cases like this from natural language, where, for example, an adverb is used. Of course, it would also possible to add an extra attribute to the type for completion time. This decision must be made by the domain designer.

UU 2.1 completes the CPS acts, and after 3.1, when the completes are grounded, the CPS acts are generated, resulting in the corresponding changes to the CPS state.

In UU 3.1, the user proposes the adoption of a (partial) recipe for shipping the oranges, as well as that focus be placed on it. These IntActs are never grounded, and thus never result in a change in the CPS state. However, as this utterance gives a good example of the introduction of a recipe, however, we will still discuss it. The recipe

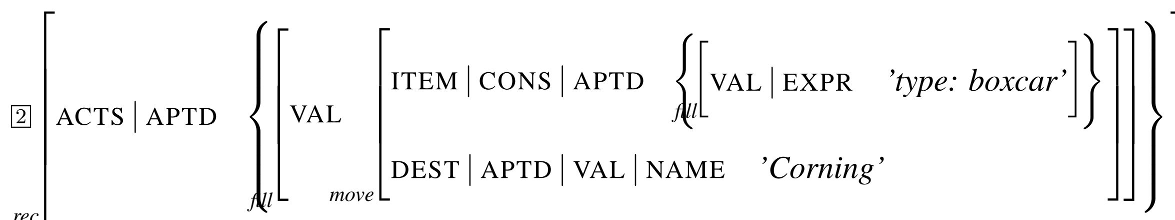


Figure 4.5: Contents of *recipe* 2

Figure 4.6: Contents of *constraint* 3

that the user attempts to introduce is shown in Figure 4.5.

This recipe consists of a single adopted objective — that of moving a boxcar to Corning. Similar to the *ship-by-train* objective above, this also has an adopted value (the destination is Corning), and a constraint on the slot of the other (that the only resources to be considered for the move should be of type *boxcar*). At this point, the recipe has no ACTION-CONSTRAINTS.

However, this recipe never makes it into the CPS state (even as something mentioned). Instead, the user decides he wants to adopt (confirm) the joint belief that there are oranges at Corning, as reflected in UU 3.2. Note that no focus change is proposed by UU 3.2. We model this in this way, as it appears that the user did not intend for further work (beyond adoption) to be done on this constraint, or on finding out the state of the world. Instead, it was intended as a quick check, but focus was intended to remain on the *ship-by-train* objective.

As discussed in the previous chapter, we model beliefs about the world state as constraints on the situation. The mentioned constraint is shown in Figure 4.6. In this thesis, we do not present a theory of constraint representation, thus we have been glossing constraints until now. The only specification we have made is that the EXPRESSION be of type *boolean*. In the case of this constraint, however, a simple gloss is not enough, as this constraint actually introduces a new embedded *resource* — the instance of the or-

anges that are at Corning. For this reason, we show this constraint as a domain-specific predicate (*at*) that takes a location and an item. It is obvious that work needs to be done on the general specification of constraints, but the representation here is sufficient for our purposes.

It is important to point out that when the *beginc-identify-constraint* is grounded in UU 4.1, the oranges instance from the constraint is also placed in the PS-OBJECTS set within the CPS state, making it available for use in further problem solving.

Discussion

Our CPS dialogue model overcomes several of the difficulties with Conversation Acts we mentioned above. First, it defines act types as well as the *content* of those acts. Second, it defines a closed set of domain-independent acts at discourse unit level (i.e., IntActs with CPS acts as arguments). Although the acts are domain-independent, they can be used with domain-specific content through PS object inheritance and instantiation). Finally, the model introduces CPS acts at the level of Argumentation Acts, which define intentions for larger chunks of discourse.

As promised above, we are now able expand our definition of communicative intention — this time to a fully-instantiated grounding act. As illustrated in the example above, we use these to represent the intended meaning of an utterance — both to do work at the grounding level (e.g., acknowledging previous IntActs) and to introduce new IntActs to be grounded. This is similar to Clark's proposed communication *tracks* [Clark1996].

4.3 Coverage of the CPS Dialogue Model

To further illustrate the use of the CPS dialogue model, and as a proof-of-concept evaluation, we explore several dialogue fragments in this section, which demonstrate dif-

ferent aspects of the dialogue model's coverage.

4.3.1 Planning and Execution

As discussed in Chapter 2, one of the things lacking in most dialogue models is the ability to distinguish between dialogue about planning and dialogue about execution. Because of this, these systems are not able to handle dialogues which include both planning *and* execution. We first show examples of the CPS dialogue model in the context of planning, then in execution, and finally in a dialogue which includes both planning and execution.

Planning

To demonstrate coverage for planning dialogue, we continue the example from Traum and Hinkelman begun in Figure 4.3. In the interest of space and clarity, we have skipped part of the dialogue (Utterance Units 5.1–13.1) which included mostly grounding interaction which has been adequately addressed by Traum and Hinkelman. We have also at times combined multiple-UU turns into a single UU, where the UUs were tagged as *conts* at the grounding level. Although modeling this is important at the grounding level, an *init* followed by *conts* are just gathered up into a single (group of) IntActs that are proposed, so at the problem-solving level, this is not an important difference.

The remainder of the dialogue is shown in Figure 4.7. With UU 13.2, the user user identifies a possible (partial) recipe for objective [1] (shipping oranges to Bath, from above). The recipe includes a single objective (action) of moving engine E1 to Dansville.[9] Note that the user does not propose to adopt this recipe for the objective,

[9]We will gloss PS objects with a boxed number, [4] in this case, and some description of their contents.

13.2 U: or, we could actually move it [Engine E1] to Dansville, to pick up the boxcar there.
init$_1$*(begin*$_1$*(c-identify-recipe(* 1 | REC | ID, 4 *[move(E1,Dansville)]))*
init$_2$*(begin*$_2$*(c-focus(*STATE | ID, 4 | ID*)))*

14.1 S: okay.
ack$_{1-2}$
init$_{3-4}$*(complete*$_{1-2}$)

15.1 U: um and hook up the boxcar to the engine, move it from Dansville to Corning, load up some oranges into the boxcar, and then move it on to Bath.
ack$_{3-4}$
init$_5$*(begin*$_3$*(c-identify-objective(* 4 | ACTS | ID, 5 *[hook(boxcar1,engine1)]))*
init$_6$*(begin*$_4$*(c-adopt-objective(* 4 | ACTS | ID, 5 | ID*)))*
init$_7$*(begin*$_5$*(c-identify-constraint(* 4 | ACONS | ID, 6 *[before(* 1 , 2 *)])))*
init$_8$*(begin*$_6$*(c-adopt-constraint(* 4 | ACONS | ID, 6 | ID*))*
init$_{9-16}$*(begin*$_{7-14}$*) [2 other actions and 2 other ordering constraints]*

16.1 S: okay.
ack$_{5-16}$
init$_{17-28}$*(complete*$_{3-14}$*)*

17.1 U: how does THAT sound?
ack$_{17-28}$
init$_{29}$*(begin*$_{15}$*(c-identify-evaluation(*FILLER(4) | EVALS | ID, 7 *[blank evaluation])))*

18.1 S: that gets us to Bath at 7 AM, and (inc) so that's no problem.
ack$_{29}$
init$_{30}$*(continue*$_{15}$*(c-identify-evaluation(*FILLER(4) | EVALS | ID, 7 *[sufficient])))*
init$_{31}$*(begin*$_{16}$*(c-adopt-evaluation(*FILLER(4) | EVALS | ID, 7 | ID*)))*

19.1 U: good.
ack$_{30-31}$
init$_{32-33}$*(complete*$_{15-16}$*)*
init$_{34}$*(begin*$_{17}$*(c-adopt-recipe(* 1 | REC | ID, 4 | ID*))*
init$_{35}$*(begin*$_{18}$*(c-defocus(*STATE | ID, 4 | ID*)))*

20.1 S: okay.
ack$_{32-35}$
init$_{36-37}$*(complete*$_{17-18}$*)*

Figure 4.7: A Planning Dialogue from [Traum and Hinkelman1992] (Continuation of Figure 4.3)

yet; he only proposes considering it as a candidate.[10] He also proposes moving the problem-solving focus to that recipe (in order to work on expanding it).

In 14.1, the system acknowledges these grounding acts and also inits IntActs to complete them. (Note that, for compactness, we use subscripted ranges to refer to series of GAs and IntActs. ack_{1-2} expands to ack_1 and ack_2, and $init_{3-4}(complete_{1-2})$ expands to $init_3(complete_1)$ and $init_4(complete_2)$.)

In 15.1, the user proposes several new actions to add to the recipe as well as ordering constraints among them. In UU 17.1, the user then asks for the system's evaluation of the recipe, which is provided in UU 18.1. The surface form of 19.1 is a bit misleading. With this "good", the user is acking the system's last utterance, and accepting the evaluation. He is also, based on this evaluation, proposing that the recipe be adopted for the objective and proposing that the focus be taken off the recipe. The system accepts these proposals in 20.1.[11]

Execution

We now give an example of a well-known execution dialogue, which we have taken from [Grosz and Sidner1986]. This is a so-called expert-apprentice dialogue, where an expert (E) guides an apprentice (A) in performing a task. The example is in Figures 4.8 and 4.9.

The context at the start of this segment is that the expert is specifying a recipe $\boxed{rec}$ to the apprentice for removing a pump. In UU 1.1, she tells the apprentice to remove the flywheel, which is modeled as not only a *c-identify-objective* and a *c-adopt-objective* as

---

[10]However, as mentioned above, the objective within it (moving the engine) *is* adopted in the ACTIONS set of the recipe. This is an example of contingent planning as discussed in [Carberry, Kazi, and Lambert1992].

[11]Again here we have modeled the final utterance as an init, which then theoretically needs to be acked by the user. We assume within the model that, if after a small pause, if there is no evidence of the user *not* having heard or understood, the inits (36–37) are automatically considered acked.

1.1 E: first you have to remove the flywheel.

$\textit{init}_{1}(\textit{begin}_{1}(\textit{c-identify-objective}(\boxed{rec} \mid \text{ACTS} \mid \text{ID}, \boxed{1}\ \textit{[remove(flywheel)]})))$

$\textit{init}_{2}(\textit{begin}_{2}(\textit{c-adopt-objective}(\boxed{rec} \mid \text{ACTS} \mid \text{ID}, \boxed{1} \mid \text{ID})))$

$\textit{init}_{3}(\textit{begin}_{3}(\textit{c-select-objective}(\boxed{rec} \mid \text{ACTS} \mid \text{ID}, \boxed{1} \mid \text{ID})))$

2.1 A: how do I remove the flywheel?

$\textit{ack}_{1-3}$

$\textit{init}_{4-6}(\textit{complete}_{1-3})$

$\textit{init}_{7}(\textit{begin}_{4}(\textit{c-focus}(\text{STATE} \mid \text{ID}, \boxed{1} \mid \text{REC} \mid \text{ID})))$

$\textit{init}_{8}(\textit{begin}_{5}(\textit{c-identify-recipe}(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2}\ \textit{[blank recipe]})))$

$\textit{init}_{9}(\textit{begin}_{6}(\textit{c-adopt-recipe}(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2} \mid \text{ID})))$

3.1 E: first, loosen the two allen head setscrews holding it to the shaft, then pull it off.

$\textit{ack}_{4-9}$

$\textit{init}_{10}(\textit{complete}_{4})$

$\textit{init}_{11}(\textit{continue}_{5}(\textit{c-identify-recipe}(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2}\ \textit{[loosen(screwsets),pull-off(wheel)]})))$

$\textit{init}_{12}(\textit{continue}_{6}(\textit{c-adopt-recipe}(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2} \mid \text{ID})))$

4.1 A: OK.

$\textit{ack}_{10-11}$

$\textit{init}_{12-13}(\textit{complete}_{5-6})$

⋮

18.1 A: the two screws are loose,

$\textit{init}_{14}(\textit{begin}_{7}(\textit{c-release-objective}(\boxed{rec} \mid \text{ACTS} \mid \text{ID}, \boxed{loosen} \mid \text{ID})))$

18.2 A: but I'm having trouble getting the wheel off.

$\textit{init}_{15}(\textit{begin}_{8}(\textit{c-select-objective}(\boxed{rec} \mid \text{ACTS} \mid \text{ID}, \boxed{pull-off} \mid \text{ID})))$

$\textit{init}_{16}(\textit{begin}_{9}(\textit{c-focus}(\text{STATE} \mid \text{ID}, \boxed{pull-off} \mid \text{REC} \mid \text{ID})))$

$\textit{init}_{17}(\textit{begin}_{10}(\textit{c-identify-recipe}(\boxed{pull-off} \mid \text{REC} \mid \text{ID}, \boxed{3}\ \textit{[blank recipe]})))$

$\textit{init}_{18}(\textit{begin}_{11}(\textit{c-adopt-recipe}(\boxed{pull-off} \mid \text{REC} \mid \text{ID}, \boxed{3} \mid \text{ID})))$

19.1 E: use the wheelpuller.

$\textit{ack}_{14-18}$

$\textit{init}_{19-21}(\textit{complete}_{7-9})$

$\textit{init}_{22}(\textit{continue}_{10}(\textit{c-identify-recipe}(\boxed{pull-off} \mid \text{REC} \mid \text{ID}, \boxed{3}\ \textit{[recipe using wheelpuller]})))$

$\textit{init}_{23}(\textit{continue}_{11}(\textit{c-adopt-recipe}(\boxed{pull-off} \mid \text{REC} \mid \text{ID}, \boxed{3} \mid \text{ID})))$

19.2 E: do you know how to use it?

$\textit{reqack}_{22-23}$

20.1 A: no.

$\textit{ack}_{19-21}$

$\textit{reqrepair}_{22-23}$

Figure 4.8: Execution Dialogue from [Grosz and Sidner1986]: Part 1

21.1 E: do you know what it looks like?
$cancel_{22-23}$
$init_{24}$(*begin*$_{12}$(*c-identify-resource*(STATE | ID, [4] *[wheelpuller]*)))
22.1 A: yes.
ack_{24}
$init_{25}$(*continue*$_{12}$)
23.1 E: show it to me please.
ack_{25}
$init_{26}$(*continue*$_{12}$)
$init_{27}$(*begin*$_{13}$(*c-identify-objective*(STATE | OBJVS | ID, [5] *[show(A,E,* [4] *)]*)))
$init_{28}$(*begin*$_{14}$(*c-adopt-objective*(STATE | OBJVS | ID, [5] | ID)))
$init_{29}$(*begin*$_{15}$(*c-select-objective*(STATE | OBJVS | ID, [5] | ID)))
24.1 A: OK.
ack_{26-29}
$init_{30}$(*continue*$_{12}$)
$init_{31-33}$(*complete*$_{13-15}$)
25.1 E: good.
ack_{30-33}
$init_{34}$(*complete*$_{12}$)
$init_{35}$(*begin*$_{16}$(*c-release-objective*(STATE | OBJVS | ID, [5] | ID)))
25.2 E: loosen the screw in the center and place the jaws around the hub of the wheel...
$init_{36}$(*continue*$_{10}$(*c-identify-recipe*([$pull{-}off$] | REC | ID, [3] *[loosen(screw),...]*)))
$init_{37}$(*continue*$_{11}$(*c-adopt-recipe*([$pull{-}off$] | REC | ID, [3] | ID)))

Figure 4.9: Execution Dialogue from [Grosz and Sidner1986]: Part 2

in the planning case, but also as a *c-select-objective*, as the expectation is that the action should be executed immediately. In 2.1, the apprentice agrees with these changes to the CPS state, but does not know a recipe for removing the flywheel. This is modeled as a *c-identify-recipe* that contains a blank recipe. The apprentice also seems willing to take whichever recipe the expert gives him, which is why there is a *c-adopt-recipe* as well.[12] The recipe content is given in 3.1 and the apprentice accepts it in 4.1.

For brevity, we have skipped part of the dialogue to UU 18.1, where the apprentice has been working for a bit and now announces that the objective of loosing the screws

[12]Note that this does not bind the apprentice to whatever recipe is given him, as he can always reject the recipe once it is given content.

has been successfully completed (*c-release-objective*). In 18.2, the apprentice notes that he is having problems with the second objective in the recipe — pulling the wheel off. This is modeled as four proposed CPS acts. First, this presupposes that the apprentice is actively trying to execute the objective, hence the *c-select-objective*. It is important to note here that the execution of the CPS act (and thus the update of the CPS state) need not, and usually does not, correspond to the point in time that the execution of the domain action actually begins. Instead, this act can be seen as one agent keeping the other agent (or actually, the CPS state) up to date about the current state of affairs (cf. [Levesque, Cohen, and Nunes1990]). The remaining three acts center around finding an appropriate recipe for the objective.

In 19.1, the expert intends to adds content to the (blank) proposed recipe, by referring to a (specific) recipe using a wheelpuller. This introduces an interesting phenomena seen when using the CPS dialogue model. Often, expressions which refer to PS objects are found — not just to resources, but also to objectives, to constraints, and as is the case here, to recipes. Note that this is a much more opaque reference to a recipe than was given in UU 3.1, where (many of) the steps and constraints were made explicit. This kind of problem-solving reference resolution presents an interesting area of future work for the interpretation and generation subsystems of an agent-based dialogue system.

In 19.2, the expert shows she is not sure if the expert will uniquely identify the recipe she refers to, and thus requests an acknowledgment. On the surface, this exchange seems that it could be modeled as a question about knowledge, which indeed it is. However, at a deeper level, this can really be seen as asking the apprentice if he knows a *recipe* that uses a wheelpuller that is appropriate for the objective at hand — which is presupposed by the recipe reference in 19.1.

Similarly, the negative response in 20.1 is modeled as a request for repair of the IntActs in 19.1. However, the expert cancels those communicative intentions, and starts establishing more basic common ground. Notice that, in the end, the recipe that was

(presumably) meant in 19.1 is eventually given in 25.2. We do not believe, however, that this should be modeled as a 6+ turn repair at the grounding level. Instead, we model it as abandoned communicative intentions at the grounding level, but a continued proposed act at the CPS level. Note that the acts in 25.2 refer to the CPS acts originally proposed by the apprentice in 18.2. These CPS acts were never rejected, and thus we have continuity at the CPS level, although we do not at the grounding level.

At this point, the expert proceeds to try to collaboratively identify (in general) the wheelpuller and only when this is established, does she attempt to specify the recipe again. In this process, note the execution of an external objective (showing the wheelpuller to the expert), in order to aid the completion of a CPS act (the identification).

Also note the use of *c-release-objective* in 25.1 to propose that the objective be considered accomplished and should no longer be pursued.

Interleaved Planning and Execution

The above examples showed how the model handles planning and execution separately. We have annotated in a like manner the dialogue from Figure 1.1 (shown in Figure 4.10). This dialogue is much richer in that it contains interleaved planning and execution.

In this example, the speakers decide to jointly pursue an objective of going to the park (UU 1.1). In 2.2, B introduces 2 possible recipes for doing so — driving or walking. To be able to plan for this objective, the participants adopt and execute another objective (looking at the weather on the internet) in order to help them in their planning decision. Finally, in 9.1 and 10.1, A and B decide to begin executing their high-level objective of going to the park.

1.1 A: let's go to the park today.
$init_1(begin_1(c\text{-}identify\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{1}\,[goto(park)]\,))$
$init_2(begin_2(c\text{-}adopt\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{1} \mid \text{ID}))$
$init_3(begin_3(c\text{-}focus(\text{START} \mid \text{ID}, \boxed{1} \mid \text{ID})))$

2.1 B: okay
$ack_{1-3} \qquad init_{4-6}(complete_{1-3})$

2.2 B: should we walk or drive?
$init_7(begin_4(c\text{-}focus(\text{START} \mid \text{ID}, \boxed{1} \mid \text{REC} \mid \text{ID})))$
$init_8(begin_5(c\text{-}identify\text{-}recipe(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2}\,[walk]))))$
$init_9(begin_6(c\text{-}identify\text{-}recipe(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{3}\,[drive])))$

3.1 A: what's the weather going to be like?
$ack_{4-9} \qquad init_{10-11}(complete_{4-6})$
$init_{12}(begin_7(c\text{-}identify\text{-}constraint(\text{START} \mid \text{CONS} \mid \text{ID}, \boxed{4}\,[weather(today,X)])))$

4.1 B: I don't know.
$ack_{10-12} \qquad init_{13}(reject_7)$

4.2 B: let's watch the weather report.
$init_{14}(begin_8(c\text{-}identify\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{5}\,[watch(report)])))$
$init_{15}(begin_9(c\text{-}adopt\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{5} \mid \text{ID})))$
$init_{16}(begin_{10}(c\text{-}select\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{5} \mid \text{ID})))$

5.1 A: no, it's not on until noon.
$ack_{13-16} \qquad init_{17}(complete_8)$
$init_{18}(begin_{11}(c\text{-}identify\text{-}evaluation(\text{START} \mid \text{FILLER}(\boxed{5}) \mid \text{EVALS} \mid \text{ID}, \boxed{6}\,[bad\ idea])))$
$init_{19}(begin_{12}(c\text{-}adopt\text{-}evaluation(\text{START} \mid \text{FILLER}(\boxed{5}) \mid \text{EVALS} \mid \text{ID}, \boxed{6} \mid \text{ID})))$
$init_{20-21}(reject_{9-10})$

5.2 A: just look on the internet.
$init_{22}(begin_{13}(c\text{-}identify\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{7}\,[look(internet)])))$
$init_{23}(begin_{14}(c\text{-}adopt\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{7} \mid \text{ID})))$
$init_{24}(begin_{15}(c\text{-}select\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{7} \mid \text{ID})))$

6.1 B: okay. [looks on internet]
$ack_{17-24} \qquad init_{25-29}(complete_{11-15})$

6.2 B: it's supposed to be sunny.
$init_{30}(begin_{16}(c\text{-}release\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{7} \mid \text{ID})))$
$init_{31}(begin_{17}(c\text{-}identify\text{-}constraint(\text{START} \mid \text{CONS} \mid \text{ID}, \boxed{8}\,[weather(today,sunny)])))$
$init_{32}(begin_{18}(c\text{-}adopt\text{-}constraint(\text{START} \mid \text{CONS} \mid \text{ID}, \boxed{8} \mid \text{ID})))$

7.1 A: then let's walk.
$ack_{25-32} \qquad init_{33-35}(complete_{16-18})$
$init_{36}(begin_{19}(c\text{-}adopt\text{-}recipe(\boxed{1} \mid \text{REC} \mid \text{ID}, \boxed{2}\ id.)))$

8.1 B: okay.
$ack_{33-36} \qquad init_{37}(complete_{19})$

9.1 A: do you want to go now?
$ack_{37} \qquad init_{38}(begin_{20}(c\text{-}select\text{-}objective(\text{START} \mid \text{OBJVS} \mid \text{ID}, \boxed{1} \mid \text{ID})))$

10.1 B: sure.
$ack_{38} \qquad init_{39}(complete_{20})$

Figure 4.10: A Planning and Execution Dialogue (from Figure 1.1)

4.3.2 Collaboration Paradigms

We also mentioned in Chapter 2 that most dialogue systems are not able to handle the full range of collaboration paradigms, or the respective roles and authority of each participant during the dialogue. In fact, most dialogue models only handle one type of collaboration paradigm — master-slave, where one participant has all authority for decision-making.

The CPS dialogue model does not explicitly model the collaboration paradigm in use — this needs to be part of the dialogue manager, which decides what to do and say at each point of the dialogue. However, the CPS model is general enough to be able to describe dialogues from most collaboration paradigms, as it represents dialogue as negotiation of changes to the CPS state.

A good example of a mixed-initiative dialogue is the dialogue in Figure 4.10, in which A and B are more or less equal. For example, in UU 5.1, A rejects B's proposal to find out the weather by watching TV and gives a reason for the rejection.

At the same time, agents are not *required* by the model to use *reject*. The expert-apprentice dialogue in Figures 4.8 and 4.9 is a good example of a dialogue that is not mixed-initiative, as the expert has quite a bit more authority.

4.4 Conclusions and Future Work

In this chapter, we have presented a novel dialogue model which is able to account for a wide range of phenomena needed for agent-based dialogue systems. The model uses interaction acts from the collaborative problem-solving model, together with a well-known theory of grounding to describe communicative intentions for utterances.

As was discussed in the chapter, there is still much work to be done in this area. First, we have mentioned several areas in which the model needs to be expanded. For example, we are still lacking a good theory of descriptions of evaluations and con-

straints. Also, the model does not take misunderstanding and recovery into account — where participants must recognize and then repair conflicts in their private understanding of the dialogue model (cf. [McRoy1998]).

Another interesting area of future work is the possible expansion of this model to be compatible with that proposed by Grosz and Sidner [1986]. Their dialogue model contains three separate but interrelated components: linguistic structure, intentional structure, and attentional structure. Linguistic structure segments utterances into discourse segments, while intentional structure describes the purposes of segments and their relations to one another. Finally, attentional structure keeps track of entities of various salience in the discourse.

We believe our model may be compatible with that of Grosz and Sidner. Linguistic structure can be partially derived from which utterances contribute to a CPS act. Intentional structure is partially recorded in the CPS state (e.g., a recipe being subordinate to its objective). We also keep track of attentional structure at the problem-solving level through focus (although it remains to be seen how this corresponds to linguistic attention and salience).

Another direction of needed future work is annotation and large-scale evaluation. Our evaluation of the model has thus far been limited to annotating dialogues in order to show the model's range of coverage. It is an open question if this kind of information can be reliably annotated by humans on a large corpus.

Another big challenge will be to create a recognizer that can automatically recognize communicative intentions (instantiated GAs) from natural language. As mentioned in Chapter 1, intention recognition algorithms use, at their core, plan recognition. In the following chapters, we discuss work on speeding up plan recognition in order to quickly support the type of recognition we will need to support the CPS dialogue model.

5 Plan Recognition: Background

In Chapter 1, we outlined several key areas in which progress must be to support agent-based dialogue systems. First we mentioned that we needed a dialogue model as well as a way of describing the communicative intentions associated with utterances. We have presented solutions to both of these in Chapter 4. Once we have a representation of communicative intentions for utterances, we need a way of performing *intention recognition*: the recognition of communicative intentions based on context and the speaker's utterance.

Unfortunately, a full model for intention recognition in agent-based dialogue is beyond the scope of this thesis. Instead, in the remaining chapters, we make several contributions to the more general field of plan recognition which we believe are the first steps towards creating a practical intention recognizer for agent-based dialogue.

Although much work has been done in intention recognition (see [Carberry1990b; Lochbaum, Grosz, and Sidner2000]) these methods assume a plan-based model of dialogue and are not directly applicable to our agent-based model. We do believe, however, they can be extended to our model, so we do not discount them. Instead, we focus on several problems with intention recognition and the more general problem of plan recognition. As plan recognition is a more general form of intention recognition, solutions in the general domain will be applicable to current intention recognizers as well

as future work on intention recognition for agent-based dialogue.

In this chapter, we first discuss the relationship between intention recognition and plan recognition. We then outline some general requirements for plan recognition and discuss previous work in this field. Finally, we conclude and introduce the solutions presented in the coming chapters.

5.1 Intention Recognition and Plan Recognition

Intention recognition is a special case of *plan recognition*: the general task of inferring an agent's goals and plans based on observed actions. In intention recognition, observed actions are speaker utterances and the goals are the speaker's communicative intentions.

Plan recognition is typically divided into two types. In *keyhole recognition*, the agent being observed is unaware of (or does not care about) the observation. In *intended recognition*, on the other hand, the agent knows it is being observed and chooses its actions in a way such to make its plan clear to the observer.[1] Intention recognition is a type of *intended recognition*, as the speaker forms his actions (utterances) in such a way to make his communicative intentions clear to the hearer.[2]

Despite the fact that intention recognition is a type of intended recognition, and that the speaker forms his utterances intentions so as to make his communicative intentions "easy" to recognize, intention recognition remains a hard problem for the community, both in terms of domain-independence, as well as runtime efficiency. All intention recognizers that we are aware of use at their core a plan recognizer. Thus any problems

[1] A third type of plan recognition occurs when the agent is trying to thwart recognition of its plans. Pollack [1986] calls this an *actively non-cooperating actor*. Very little research has been done for this third type of recognition (although cf. [Azarewicz et al.1986]), which may be why it is frequently not included in the typology.

[2] Note, that this is the case even in deceptive conversation, as the speaker forms his utterances so as to make his feigned intentions clear.

with plan recognizers in general have been inherited also by intention recognizers. We now turn our attention to plan recognition in general.

5.2 Requirements for Plan Recognition

Plan recognition has not only been used in dialogue systems, but also in a number of other applications, including including intelligent user interfaces [Bauer and Paul1993; Horvitz and Paek1999; Rich, Sidner, and Lesh2001], traffic monitoring [Pynadath and Wellman1995], and hacker intrusion detection [Geib and Goldman2001]. All of these applications (including dialogue) have a common set of requirements they place on a plan recognizer:

1. **Speed:** Most applications use plan recognition “online,” meaning they use recognition results before the observed agent has completed its activity. Ideally, plan recognition should take a fraction of the time it takes for the observed agent to execute its next action.

2. **Early prediction:** In a similar vein, applications need accurate plan prediction as early as possible in the observed agent’s task execution. Even if a recognizer is fast computationally, if it is unable to predict the plan until after it has seen the last action in the agent’s task, it will not be suitable for online applications, which need recognition results *during* task execution.

3. **Partial prediction:** If full recognition is not immediately available, applications can often make use of partial information. For example, if the parameter values are not known, just knowing the goal schema may be enough for an application to notice that a hacker is trying to break into a network.

As we discuss below, previous work in plan recognition does not provide these needed features. Typically, systems will sacrifice one attribute for another.

5.3 Previous Work in Plan Recognition

In this section, we discuss previous work in plan recognition. This can be divided into two different types. The first is plan recognition based on logic, while the second includes probabilities.

5.3.1 Logic-based Plan Recognition

Most plan recognizers use a plan library, which represents goals in the domain, and the (typically hierarchical) plans associated with them. Logic-based recognizers can be characterized by the use of logical methods to exclude goals and plans in the hierarchy made impossible given the observed actions.

There have been several types of logic-based plan recognizers. We first discuss work that bases plan recognition on chaining. Then we discuss plan recognition as circumscription, and finally, plan recognition based on parsing algorithms.

Plan Recognition as Chaining

Allen and Perrault [1980] created one of the earliest plan recognizers. Given a single observed action, the recognizer used various rules to either forward chain from the action to a goal, or backwards chain from an expected goal to the action. Rules supported not only chaining on preconditions and effects, but also hierarchically to higher levels of recipes. Heuristics were used to control and focus rule application for chaining.

Carberry [1983; 1990b] extended Allen and Perrault's work to cover multiple successive action observations. Each new action is independently upwards chained until further chaining would create ambiguity. Then, the new action is merged into the plan recognized so far based on previous observations. Ambiguity of where a plan "attaches" is resolved by the use of focusing heuristics, which assume that action observations are often coherently clustered together.

Plan Recognition as Circumscription

The seminal work on plan recognition was done by Kautz [1987; 1990; 1991][Kautz and Allen1986], who casts plan recognition as the logical inference process of circumscription. This provided a rich plan representation — essentially that of first order logic and a temporal logic to represent actions and time.

Kautz represented the space of possible plans as a plan library called the event hierarchy, which included both abstraction and decomposition (subaction) relations. Goals and actions were represented as complex schemas that included parameter values. Certain actions were labeled as *end* actions, meaning that they were an end unto themselves, or a possible ultimate goal of an agent.

Kautz showed that by assuming that the event hierarchy is complete and that all events are disjoint, plan recognition becomes a problem of logical circumscription. Given a certain set of observations (also represented in first order logic) a set of *covering models* is computed which are somewhat like possible worlds in which the observations are true, and each contains a separate possible goal and plan for the agent.

Runtime of the recognizer is exponential in the size of the event hierarchy (e.g., all goals and subgoals), which means it is not scalable to larger, more realistic domains. However, it does have several other features, including a rich representational power (including interleaved plans, partially-ordered recipes, and goal and action parameters, to name a few). It also supports partial prediction through the ability to predict just a goal schema as well as to predict an abstract goal. As discussed below, it also suffered from the general inability of logic-based systems to deal with ambiguity.

Plan Recognition as Parsing

To make plan recognition more tractable, Vilain [1990] describes a method of converting a subset of Kautz's plan hierarchy into a grammar. Plan recognition is then performed by running a chart parser over observed actions. By using this approach,

runtime complexity becomes $O(|H|^2n^3)$ where H is the set of goals and subgoals in the plan library and n is the number of observed actions.

This vast improvement over exponential runtime comes at a cost: the grammar approach decreases the representational power substantially; it requires totally-ordered recipes; and does not handle goal and action parameters. (Vilain suggests that parameters could be handled as a feature grammar, although this would make the algorithm NP-Complete.) The lack of goal parameters means a possible explosion in the number of goals in certain domains, since each instance of a goal schema must be modeled as a separate goal.

In addition, it is not entirely clear if online predictions can be made by the recognizer. Vilain suggests that this could be done by looking at dotted rules on the chart, but it is not clear how much predictive power this would give the recognizer.

General Shortcomings of Logic-based Recognizers

A general problem with logic-based recognizers (as noted in [Charniak and Goldman1993]) is their inability to deal with ambiguity. This comes from the fact that, upon each new observed action, they prune away only the predictions which become logically impossible. Unfortunately, this impacts early prediction substantially, as most plan recognition domains are highly ambiguous, especially when only the first few actions from a plan have been observed. In order to disambiguate further, uncertain reasoning is often used.

5.3.2 Probabilistic Plan Recognition

Several lines of probabilistic plan recognition have been explored. We discuss here the use of Dempster-Shafer theory, probabilistic abduction and belief networks.

Dempster-Shafer Theory

Two systems use Dempster-Shafer theory (DST) to add probabilistic reasoning to plan recognition. Carberry [1990a] used DST to her logic-based recognizer (see above) to do default inferencing when further upwards chaining was ambiguous.

Bauer [1995] uses DST to represent and combine the probability of goals given observed actions. He uses a subset of Kautz' plan library which includes an abstraction hierarchy and a single level partially ordered recipes. He uses results of previous recognition sessions to learn a DST *basic probability assignment* (bpa) which roughly corresponds to the a priori goal probability (an abstract goal is defined a the set of its base goals).

In addition, he uses the plan library itself (and a corpus if available [Bauer1994]) to train another set of bpas which roughly correspond to the probability of a goal given an observed action. A bpa is defined for each action in the domain, and gives probability mass to each goal in which it is part of a recipe.

The recognition algorithm is as follows: the prediction bpa is initialized to the a priori goal probabilities. Then, for each observed action, the precomputed bpa for that action is retrieved, and then each of the possible goals is logically checked with respect to constraints (e.g., ordering constraints). If all constraints for all goals hold, the bpa remains the same. Otherwise, probability mass is taken away from logically impossible goals and redistributed. Then, this bpa is combined with the prediction bpa by Dempster's rule of combination resulting in the new prediction bpa.

It is unclear if this algorithm is scalable, however. DST is known to be exponential in the general case, and although Bauer mentions some possible solutions (like restricting bpa subsets to be only abstract goals) it is unclear how this restriction would be handled by Dempster's rule of combination and what the effect on recognition would be. Also, the approach does not support a decomposition hierarchy, and thus is unable to make predictions about intermediate subgoals and plans.

Probabilistic Abduction

Appelt and Pollack [1991] designed a framework in which plan recognition[3] could be modeled as weighted abduction. The framework allows inferences to be encoded as prolog-like rules with a weight attached to them. If the consequent of the rule can be logically proven, there is no cost. However, if it is assumed, then the algorithm incurs the cost of the weight of that step. Out of all possible solutions, the one with the lowest weight is then chosen.

Appelt and Pollack mention several drawbacks to their work. First, in the general case, the algorithm is intractable (NP-hard). Also, weights assigned to abduction rules are not probabilities and must be assigned by hand. They report that local changes in these rules can affect global recognition in subtle ways.

Goldman et al. [1999] also model plan recognition as an abduction problem. They model the process of plan execution, and then reverse the decisions to make an abductive model. In addition, three parts of the execution process are made probabilistic: the agent's choice of a top goal, the agent's choice between competing recipes for a goal (or subgoal), and the agent's choice of what action to execute next (from the set of currently executable actions).

This framework is the first of which we are aware to model plan recognition with the fact in mind that the agent is executing the action, as opposed to other work which just works on a plan library data structure. Because of this, they are able to model many things with other systems could not, including multiple, interleaved plans and evidence from *failure* to observe an action.

Like Appelt and Pollack, however, Goldman et al. define a theoretical framework, but do not deal with the problem of tractability. Although they do not analyze complex-

[3] Actually, they do what they call *plan ascription*, which is the (more difficult) process of attributing mental states to an agent, the combination of which can then signal that the agent has a mental plan of the form described in [Pollack1986].

ity, it is likely that this framework suffers from the same intractability problems that Appelt and Pollack's abduction framework had.

Belief Networks

Charniak and Goldman [1993] use a belief network (BN) to encode the plan recognition problem. Nodes in their BN include propositions such as the existence of an object or event, its type, and its role within some plan. As actions are observed, they are added to the network in this kind of encoding (with the appropriate arcs between them), and new nodes are generated which explain possible connections between them and the possible plans encoded in the network. After these nodes (and connections) have been added, the posteriori probabilities of other nodes (especially goals) can be computed to predict the plan.

Huber et al. [1994] propose a method to automatically convert a plan execution library into a BN, albeit one with a different structure. Their BNs include only events (not parameters) and directly encode the links between them (not through intermediary role nodes like Charniak and Goldman).

Unfortunately, reasoning with BNs is exponential in the size of the network. To attempt to deal with this, Charniak and Goldman use a message-passing algorithm to keep the number of nodes restricted, although the size of the network grows with each new observation (and the likely goals chained from it). The system of Huber et al. has a static BN and is likely not scalable to large plan libraries.

5.4 Goal Recognition

In the last section, we discussed previous work in plan recognition. We now discuss work on a special case of plan recognition: *goal recognition*. Whereas the task of plan

recognition is the recognition of an agent's goal and plan, goal recognition attempts only to recognize the goal.

Although not as informative as full plan recognition, goal recognition has been an active research area of late, partially because it has been noticed that many applications simply do not need full plan recognition results. For example, Horvitz and Paek [1999] built an AI receptionist which observed actions (including natural language utterances) to determine the user's goal, which the receptionist then did for them. Here, the goal was something only the receptionist itself could accomplish, thus the users typically did not have a plan.

Additionally, goal recognition naturally removes some of the ambiguity present in plan recognition. It is still the case that a set of observed actions could be accounted for by any number of goals, but plan recognition has the additional ambiguity that, even if the agent's goal can be unambiguously identified, it could be associated with a large number of plans, all consistent with the observed actions. For this reason, in fact, most of the plan recognizers mentioned above do not predict a fully-specified, fully-disambiguated plan at each timestep, but rather a *partial* plan that includes only those parts which are disambiguated. We believe that a fast goal recognizer could be used in a hybrid system to focus the search in a slower plan recognizer (although we leave this to future work).

Goal recognizers can be classified by the goal structure they try to recognize. *Flat* goal recognizers attempt to recognize goals at just one level, typically the top-level goal. *Hierarchical* goal recognizers, on the other hand, attempt to recognize active subgoals in addition to the top-level goal. Note that hierarchical goal recognition is different from general plan recognition in that in plan recognition, the attempt is to recognize the entire plan tree, whereas with hierarchical goal recognition, one only attempts to recognize the chain of the active subgoals, i.e., the line of subgoals which trace the last observed action to the top-level goal.

We first discuss previous work on flat goal recognizers, and then hierarchical goal

recognizers.

5.4.1 Flat Goal Recognizers

Logic-based Systems

Following recent successful work on using graph analysis in doing planning synthesis [Blum and Furst1997], Hong [2001] uses graph analysis for goal recognition. His system incrementally constructs a *goal graph* consisting of nodes representing state predicates and observed actions. Each observed action has incoming edges from state predicates that fulfill its preconditions, and outgoing edges to predicates that are its effects. Predicates which remain true across actions are also connected. Predicates also connect to goal nodes whose goal state they contribute to. This provides a list of all goal states partially or fully fulfilled by the actions up until the last observation. The algorithm then uses the graph to compute which goals were causally linked to which actions. If a majority of observed actions contributed to a certain goal, it is reported as a recognized goal.

The algorithm does not require a hand-built plan library, but rather just uses descriptions of base-level actions and high-level goal states. As Hong points out, however, this algorithm is only appropriate for post hoc goal analysis, and not online goal recognition, as it does not quickly converge on a single goal. The reason for this is that the effects of an action may contribute to any number of goals, and it only becomes clear near the end of the agent's execution which of these is really being focused on.

Lesh's RIGS-L system [Lesh and Etzioni1995b; Lesh and Etzioni1995a; Lesh and Etzioni1996; Lesh1998] uses analysis of a different kind of graph to do goal recognition. RIGS is initialized with a fully-connected *consistency graph* of action and goal schemas and instantiated actions observed thus far. Edges between action schema nodes are used to signify *support* between them, and edges to goal schema nodes signify *completion*. Given this graph, the algorithm uses rules to remove graph elements while

still keeping the graph correct. For example, the *matching* rule removes an edge $e_{x,y}$ where no effect of x matches a precontidion of y (and thus does not directly support it). The *goal connection* rule deletes goal schemas which are no longer connected to the graph. After the algorithm has run, any goal schema that is not connected is no longer consistent with the evidence, and any remaining goal schemas are instantiated by the algorithm and predicted as possible goals.

The runtime complexity of RIGS-L is $O(|G| + (|A| + |L|)^6)$ where G is the set of goal schemas, A is the set of action schemas, and L is the set of observed actions. Note that, although this is linear in the number of goal schemas, it is only polynomial overall, unless $|G| >> |A|$, which we do not believe is the case in most domains.

Lesh then uses RIGS-L as a component of the BOCES goal recognizer, which uses *version spaces* from the machine learning field to represent the set of possible goals and mark which are consistent (without, however, actually *enumerating* the goals). The set of goals are defined and then based on this definition, the goal recognizer keeps track of boundaries between those goals which are consistent and which are not. Lesh shows that BOCES has a runtime complexity of $O(log(|G|))$ for a certain subclass of goals called *decomposable goals*, goals in which adding a conjunct makes them more specific (like searching for an item with a set of features). Runtime for other classes of goals is the same as that of RIGS-L.

For decomposable goals, BOCES has been shown to run quickly for even hundreds of thousands of goals. However, these goals are defined in a certain way, namely the combination of conjuncted domain predicates, which is typically the case in decomposable goals such as constrained searching. However, many typical goal recognition domains do not exclusively include decomposable goals. For decomposable goals, however, BOCES is probably unbeatable.

Logic-based goal recognizers in general also have the same drawbacks mentioned for logical plan recognizers above, namely, that they are unable to distinguish between logically consistent goals, which leads us to probabilistic flat goal recognizers.

Probabilistic Systems

Horvitz and Paek [1999] use a 3-layered Belief Network to recognize users' goals in a secretarial setting. The system not only uses observed actions in the network, but also other factors like world state. The top layer network tries to recognize an abstract goal. When confidence in a single goal at this level is high enough, control passes to the next level, which attempts to recognize a more concrete goal, and so on. The system is able to perform partial recognition because it can return just an abstract goal when it is not certain enough about a more specific version. As the system uses a Belief Network, its worst-case complexity is exponential in the size of the network. Also, as is the case for probabilistic systems, probability distributions must somehow be estimated for each of the nodes and it is unclear this would be done.

Albrecht et al. [1998] use a dynamic belief network (DBN) to predict the top-level goal and next action in a multi-user dungeon (MUD) game. They estimate probabilities from logs of actual game sessions, where a user attempts to complete one of 20 quests (goals). Although not reported, the runtime complexity of the recognizer appears to be linear in the number of goals, and is quite similar to the statistical goal schema recognizer we present in Chapter 7 (although see Section 7.2.1 for a discussion of differences). Their recognizer, however only recognizes atomic goals and is not able to handle parameters. It also does not support partial prediction. However, it was the first goal recognizer of which we are aware which used a large corpus to learn probabilities as well as to evaluate the recognizer.

5.4.2 Hierarchical Goal Recognizers

The last section discussed flat goal recognizers, which only recognize the agent's top-level goal. In this section, we report on several recent recognizers which recognize all of an agent's active subgoals, as well as the top-level goal.

Pynadath [1999][Pynadath and Wellman2000] uses probabilistic state-dependent

grammars (PSDGs) to do plan recognition. PSDGs are probabilistic context-free grammars (PCFGs) in which the probability of a production is a function of the current state. This allows, for example, the probability of a recipe (production) to become zero if one of its preconditions does not hold. Subgoals are modeled as non-terminals in the grammar, and recipes are productions which map those non-terminals into an ordered list of non-terminals or terminals. During recognition, the recognizer keeps track of only the current productions and the state variables as a DBN with a special update algorithm. The most likely string of current productions is predicted as the current hierarchical goal structure.

If the total state is observable, Pynadath claims the complexity of the update algorithm to be linear in the size of the plan hierarchy (number of productions).[4] However, if the state is only partially observable, the runtime complexity is quadratic in the number of states consistent with observation, which grows exponentially with the number of unobservable state nodes.

Additionally, the recognizer only recognizes atomic goals and does not take parameters into account. Finally, although the PSDG allows fine probability differences for productions depending on the state, it is unclear how such probability functions could be learned from a corpus, as the state space can be quite large.

Bui [2002][Bui, Venkatesh, and West2002] performs hierarchical recognition of Markov Decision Processes. He models these using an Abstract Hidden Markov Model (AHMM) which are multi-level Hidden Markov Models where a policy at a higher level transfers control to a lower level until the lower level 'terminates.' The addition of memory to these models [Bui2003] makes them very similar to the PSDGs used by Pynadath in that each policy invokes a 'recipe' of lower-level policy and does not continue until the lower level terminates.

Recognition is done using a DBN, but because this is intractable, Bui uses a method called Rao-Blackwellization (RB) to split network variables into two groups. The first

[4]This claim is disputed in [Bui2002].

group (which includes the state variables as well as a variable which describes the highest terminating state in the hierarchy) is estimated using sampling methods. Then, using those estimates, exact inference is performed on the second part (the policy variables). The separation is such that exact inference on the second group becomes tractable, given that the first group is known.

The recognizer was used in a system which tracked human behavior in an office building at three abstract levels, representing individual offices at the bottom level, then office groups, then finally the entire building. Policies at each level were defined specific to each region (for example the policy (behavior) of using the printer in the printer room). In this model, only certain policies are valid in a given state (location), which helps reduce the ambiguity. Typically, the domain is modeled such that lower-level policies become impossible as the agent moves to another room, which makes it fairly clear when they then terminate.

Although the algorithm was successful for this tracking task, it is unclear, however, how effective estimation of policy termination would be in general (e.g., when most policies are valid in most states). Also, similar to Pynadath, this method only recognizes atomic goals and does not support parameters.

5.5 Towards Statistical Goal Recognition

As mentioned above, we need goal recognizers which are fast, and make early (and possibly partial) predictions. However, most current recognizers are either not scalable or severely limit the representation of the domain.

In the following chapters, we present a *statistical goal recognizer* which uses machine learning techniques to train the recognizer on a particular domain given a corpus. As it learns domain behavior from the corpus, it does not utilize a plan library and does not therefore limit plan representation in that respect. In addition, it supports parameterized goal and action schemas and can make partial predictions if not all parameter

values are known. We will show that it is scalable and can make quick and early predictions.

The remainder of the thesis is as follows. As the recognizer needs a corpus to be trained on, in Chapter 6 we present the two corpora which we use in our experiments. The first was gathered from human users in the Linux domain. However, as many domains do not lend themselves to easy observation, we present a general method for stochastically producing artificial corpora for plan recognition and use this method to produce a corpus in the emergency planning domain.

In Chapter 7, we present a flat goal recognizer which is linear in the number of goals and present its performance on the two corpora described above. Finally, in Chapter 8, we extend this flat recognizer into a hierarchical goal recognizer and present experimental results for it as well.

Finally, in Chapter 9, we conclude the thesis and discuss directions of future work.

6 Obtaining Corpora for Statistical Goal Recognition

Over the past 10+ years, many fields in AI have started to employ corpus-based machine learning techniques. Plan recognition, however, seems to have lagged behind. We are only aware of a few plan/goal recognizers [Bauer1996a; Albrecht, Zukerman, and Nicholson1998; Blaylock and Allen2003; Blaylock and Allen2004] (one of which is our own) that are trained on corpora. We believe a major reason for this is the lack of appropriate corpora for plan recognition (which we will term *plan corpora*).

It is not that the field could not make use of plan corpora. Besides the machine-learning based systems mentioned above, many of the plan/goal recognizers described in Chapter 5 make use of probabilities, but only briefly mention (if at all) how such probabilities could be learned.[1] In addition to providing training data, corpora could also be used to evaluate the performance of recognizers, or even as benchmarks to compare performance across recognizers (something which, as far as we are aware, has never been done).

In this chapter,[2] we describe our efforts in creating two plan corpora — the Linux corpus and the Monroe corpus — which are used in later chapters to train and test our goal recognizer. Associated with the latter, we also introduce a novel method for

[1] A notable exception is Bauer [1994].

[2] Some contents of this chapter were reported in [Blaylock and Allen2005b].

artificially creating plan corpora.

The remainder of the chapter is as follows: In Section 6.1, we introduce a terminology for describing plan corpora. In Section 6.2, we describe previous work in the creation of plan corpora. Then in Section 6.3, we describe the creation of the Linux corpus from human users. In many domains, data collection from humans may be difficult. Section 6.4 describes several challenges for gathering plan corpora based on human data. As an alternative to data collection from humans, Section 6.5 introduces a general method for stochastically creating artificial plan corpora. In Section 6.6, we describe our use of this method in creating the Monroe corpus. In Section 6.7, we discuss the advantages and disadvantages of human and artificial corpora and finally, in Section 6.8, we conclude and mention future work.

6.1 Definitions

We briefly present a few definitions that will be used in the next few chapters. We define a *plan session* to be a single session in which an agent is observed. A *plan corpus* is a collection of plan sessions which are minimally annotated with the sequence of actions observed. A *goal-labeled plan corpus* is a plan corpus in which action sequences are labeled with the agent's top-level goal(s), and a *plan-labeled plan corpus* is additionally labeled with the agent's plan(s). Note that a corpus labeled with hierarchical goal information is nearly equivalent to a plan-labeled corpus, so we do not include it in our taxonomy.

6.2 Existing Plan Corpora

In this section, we mention some previously gathered plan corpora. We first present corpora that are unlabeled, and then goal-labeled corpora. We are not aware of the

existence of any plan-labeled corpora[3] besides the Monroe corpus which we introduce in Section 6.6.

6.2.1 Unlabeled Data

Several projects in ubiquitous computing [Ashbrook and Starner2003; Patterson et al.2003] have gathered raw data of a user's state over time (location and speed from GPS data) which they use to predict user activity. However, this data is not directly usable by most plan/goal recognizers which expect a sequence of actions as input.

Davison and Hirsh [1997; 1998] collected a corpus of over 168,000 Unix commands from 77 users over a period of 2-6 months. The corpus consists of timestamped sequences of commands (stripped of arguments) as recorded by the history mechanism of tcsh. They then use this for training and testing algorithms for next command prediction.

It is unclear how useful such data would be by itself for plan recognition (although Bauer [1998] has done work on using such to automatically construct recipe libraries).

6.2.2 Goal-labeled Data

The MUD Corpus

Albrecht et al. [1998] extracted a plan corpus from logs from a Multi-User Dungeon (MUD) game (which we will term the *MUD corpus*). A single log includes a sequence of both player location (within the game world) as well as the sequence of commands executed in the session. In addition, the MUD records each successful quest completion, which is used to automatically tag plan sessions with a top-level goal. The corpus

[3] Although Bauer [1996b; 1996a] introduces a method for creating such. This is described in more detail in Section 6.4.2.

consists of 20 quests, 4,700 locations, and 7,200 observed actions. This corpus was used to train and test the flat goal recognizer described in Section 5.4.1.

Albrecht et al. report that the corpus data is quite noisy: First because of player errors and typos, and also because players in MUDs often interleave social interaction and other activities. It is also important to note that the goals in the corpus are atomic, as opposed to being parameterized goal schemas.

The Unix Corpus

Lesh [1998] created a goal-labeled corpus in a more controlled setting. He gathered the Unix corpus using test subjects (users) at the University of Washington Department of Computer Science. Users were given an English description of a task (goal) (such as "Find a file that contains the word 'motivating'") which they then attempted to complete using a subset of Unix commands.[4] The users then indicated success or failure by a special command which terminated the plan session. Lesh then post-processed the corpus by converting raw command strings into a parameterized action representation for Unix commands. The corpus is rather small and contains only 59 successful sessions with 11 distinct goals.[5]

6.3 The Linux Corpus

In Chapter 7, we will present a flat goal recognizer that can recognize both a goal schema and its parameter values, which is trained on a goal-labeled corpus. Because the MUD corpus does not include parameterized goal schemas, it was not ideal for training and testing the recognizer. The Unix corpus, on the other hand, does include

---

[4] Subjects were not allowed to use a number of constructs such as pipes or scripting languages like awk.

[5] Although Lesh used these as parameterized goal schemas, the same parameters were used for each goal each time, so these could be thought of as simply 11 distinct atomic goals as well.

goal schemas and parameters, but was very small. We created the Linux corpus as an extension to the Unix corpus.

The goals of the Linux corpus collection were threefold. First, we wanted to increase the size (i.e., number of plan sessions) of the Unix corpus. Second, we wanted to increase the complexity of the recognition task by adding more goal schemas. And finally, we wanted to increase the variety of goals by allowing multiple parameter values for each goal schema.

We first describe how the Linux corpus was gathered and then how it was post-processed. We then make some general observations on the resulting corpus.

6.3.1 Data Collection

Data for the Linux corpus was gathered from volunteer students, faculty and staff in the University of Rochester's Department of Computer Science. Volunteers were instructed to run a program installed on the local network, which led them through a collection session. This was advantageous, as it required no human supervision and could be run by users at their own convenience. Multiple concurrent users were also supported. Users were able to run the script as many times as they wished, in order to contribute more plan sessions.

User-Specific Data

On a first run by a user, the program gathered general data about him (as detailed below) and created an account for him. The user was then shown a set of general instructions about what the experiment was about and how the plan session should proceed. In particular, users were told that they would be given a task in Linux to perform, and that they should perform it using only the command line of the shell they were currently in. In addition, they were asked to avoid using certain constructs such as pipes or scripting

languages, as we wanted to keep the mapping of one command string to one actual domain action. The actual instructions given can be found in Appendix A.

Goal and Start State Generation

At the start of each session, a goal and start state were stochastically created. Each goal schema in the domain was given an a priori probability, and the program used these to stochastically choose a goal schema for the session. Each goal schema had associated with it a list of possible parameter values for each parameter position, and one of these was chosen randomly for each parameter position, giving us an instantiated goal.

Goals were similar to those used in the Unix corpus, including goals like "find a file that ends in '.txt'" and "find out how much filespace is free on filesystem /users." A list the goal schemas found in the Linux corpus can be found in Appendix B.

For each session, we generated a new start state — a new directory structure and the files within it.[6] A particular challenge was to ensure that the generated goal was achievable. Instead of trying to do this on a case-by-case basis given a generated goal, we decided to guarantee that any generated goal would be possible in any start state.

To do this, we first settled on a static set of files and directory names from which all start states were generated. The set was carefully coordinated with the set of goal schemas and possible parameters. For example, one of the possible instantiated goals was "delete all files which contain more than 40,000 bytes." To make this achievable, our static file set included several files which were larger than 40,000 bytes.

For a given session, we first created a small subset of the directory tree which ensured that all goals were possible. The remaining part of the tree was then generated

[6] It appears that the Unix corpus used a static start state for each session. We chose to generate random start states to avoid a possible learning effect in the corpus. Users who participated in multiple plan sessions may have learned the directory structure, which could have made certain tasks e.g., finding files, much easier. Eventually, one may want to model a user's knowledge of the environment, but we chose to leave this to future research.

randomly given the set of remaining directory names and files.

The Plan Session

Once the goal and start state were generated, the user was to be presented with the goal. We followed Lesh in presenting the goal as natural language text to the user. We associated a template with each goal schema which was instantiated by substituting variables with the values of the corresponding schema parameter values. Appendix B shows the Linux goal schemas and their corresponding English templates.

The goal was displayed to the user and he was given a shell-like prompt in which to input commands. The user's commands as well as their results (the output to `stdout` and `stderr`) were recorded and stored in the corpus. In addition, the system supported several meta-commands which were not directly recorded in the corpus:

- `success` — used to indicate that the user believes that they have successfully completed the task.
- `fail` — used to end the session without successful task completion.
- `task` — used to redisplay the session goal at any time.
- `instruct` — used to redisplay the general instructions at any time.
- `help` — used to display general help with the system.

The system continued recording commands and results until the user used the `success` or the `fail` command.

Data Recorded

For each plan session, the following data was recorded and is available in the raw version of the corpus:

- *Time*: date and time the session began.

- *User ID*: a unique number that identifies the user.
- *Linux level*: the user's reported proficiency in Linux between 1 (lowest) and 5 (highest).
- *User status*: whether the user was an undergraduate student, graduate student, or other.
- *Goal*: the instantiated goal schema for the session.
- *Goal text*: the actual text that was presented to the user.
- *Reported result*: whether the user reported success or failure for the session.
- *Directory structure*: the directory tree generated for the session (actual files used were static for each session and are also available).
- *Commands and results*: each issued command along with its result (from a merged `stdout` and `stderr` stream).

6.3.2 Corpus Post-processing

After the experiments, we performed various operations in order to transform the raw corpus into something we could use for training and testing our goal recognizer.

First, we excluded all sessions which were reported as failures, as well as sessions with no valid commands. Although such data could possibly be useful for training a recognizer to recognize goals which will not be accomplished by the user alone, we decided to leave such research for future work.

We also converted issued Linux commands into parameterized actions. Unlike actions in many domains used in plan recognition, Linux commands do not nicely map onto a simple set of schemas and parameters. To do the mapping, we defined action schemas for the 43 valid Linux command types appearing in the corpus. This allowed us to discard mistyped commands as well as many commands that resulted in errors. More details about this conversion process as well as the list of action schemas themselves can be found in Appendix C.

	Original	Post-processed
Total Sessions	547	457
Failed Sessions	86	0
Goal Schemas	19	19
Command Types	122	43
Command Instances	3530	2799
Ave Commands/Session	6.5	6.1

Table 6.1: Contents of the Linux Corpus

Table 6.1 gives a comparison of the original and post-processed versions of the corpus.

The post-processed corpus had 90 less plan sessions (86 failed and 4 where the user reported success but did not execute any successful commands!) The drastic reduction in command types (from 122 to 43) is mostly due to mistyped commands which either did not exist or which were not the intended command (and therefore not used with the right parameters).[7] The removal of unsuccessful commands was the main contributor to the drop in average commands per session.

6.3.3 General Comments

As discussed above, the Linux corpus was gathered semi-automatically from humans. As a consequence, it contains mistakes. A frequent mistake was typographical errors. The post-processing step described above helped ameliorate this somewhat — as it was able to detect incorrectly typed commands (at least in cases where the mistyped command wasn't also a successful command). However, it only checked the command itself, and not its parameters. This lead to cases of the user using unintended parameters (e.g., `ls flie` instead of `ls file`).

[7] A frequent example was using the command `ld` instead of the (supposedly) intended `ls`.

Another phenomenon that we were not able to automatically detect was the user's lack of knowledge about commands. For example, one user, upon getting the task of finding a file with a certain name tried several times in vain to use the command `grep` to do so, where the command he was likely looking for was `find`.[8]

Finally, another source of noise in the corpus is that the users themselves reported whether they had accomplished the task successfully. We have seen several cases in the corpus where a user apparently misunderstood the task and reported success where he had actually failed. Overall, however, this does not appear to have happened very often.

6.4 General Challenges for Plan Corpora Collection

As discussed above, several plan corpora have been created. As we discuss in this section, however, there remain significant challenges to making more corpora available. To highlight these, we first present a basic model of corpus collection, using natural language corpora as an example.

A corpus can be seen as consisting of two general parts: (1) one or more sequences of *base data*, and (2) *annotations* on that data. The first is what is required for a minimal unlabeled corpus, while the annotations can provide labeling of various sorts. Prototypically, a natural language corpus has base data which is a sequence of words. Many different annotations can then be built on top of this (using the words as building blocks), including parse trees, dialogue acts, and so forth. Similarly, a plan corpus can be seen as having base data which is a sequence of actions which can then be annotated, e.g., with plans and high-level goals.

It is important to note that the base data is often not directly observable. For natural language texts, base words are often directly available,[9] but this is not the case for speech. In this case the raw data (the speech signal) must be converted into the base

[8]The command `grep` is used to find text in a file or set of files, not to find a file in a directory tree.

[9]Although this is not the case for languages such as Japanese which do not break words in text.

data (typically done by human transcription, although speech recognition can be used as well).

Based on this classification, we can now divide the challenges for gathering new plan corpora into two categories: getting the base data (e.g., the unlabeled corpus) and then getting various labels for it. We discuss each of these challenges in turn.

6.4.1 Getting Base Data

We have now discussed several types of existing plan corpora, including our own contribution: the Linux corpus. It is interesting to note that most of these are in the computer domain. We do not believe that this is a coincidence. These are domains were the raw observed data is very close to the form of the desired base data (e.g., the action representation). Although some processing is usually necessary (e.g., our conversion of command strings to a parameterized actions in the Linux corpus), it can be typically be automated.

In many plan recognition domains, however, this is not the case. Take for example Kautz' famous cooking domain which includes actions like `Boil` and `MakeNoodles`. Unless these are asserted in language (e.g., talking about the domain), they would likely need to be observed visually. Of course, if actions are not easily observable in corpus collection, it also means they are likely not easily observable by the plan recognizer itself, which then begs the question of why we would need such data in the first place. We believe that it is at least important to keep this point in mind, however.

6.4.2 Getting Labeled Data

Oftentimes, base data is not enough, especially for doing supervised machine learning. Often, some sort of labels on the data are needed: e.g., goals, plans, and so forth.

In the labeled corpora discussed above, the MUD corpus was in a domain with the special property that the system could notice when a top-level goal had been reached

and record it (although it was not apparent at which point the player began pursuing that goal). In many domains, however, the agent's goal is not always readily observable.

For the Unix and Linux corpora, subjects were given a top-level goal and had to report their own success or failure in achieving it. This, of course, made the top-level goal labeling easy, but it was also prone to errors in success reporting, as are described above. This kind corpus gathering is also potentially expensive, as subjects must be recruited to perform tasks.

Probably a bigger challenge is getting plan-labeled data. This is likely the most valuable type of corpus for plan recognizers, yet as mentioned above, we are not aware of the existence of such a corpus.

An obvious, yet expensive way of obtaining labeled data is manual annotation. In natural language processing, this is done frequently when no automatic algorithm can be found for the task. However, manual annotation is time-consuming and can be prone to human errors. Special care must also be taken, when the annotation task is divided among annotators, that things are annotated in the same way (cf. [Carletta et al.1997]). We are not aware of any such human annotation effort in the field of plan recognition and it is difficult to predict how difficult the task of plan corpus annotation would be. We would venture that top-level goal annotation might be fairly simple, but that hierarchical plan annotation would be much more time-consuming and error prone. However, such a hypothesis would have to be tested.

An alternative solution for plan annotation has been proposed by Bauer [1996b; 1996a], who gathered recorded user action sequences and corresponding system state) from an email program and then used a plan recognizer to label them with the appropriate goal and plan post hoc. This post hoc recognition can be much more accurate than online prediction, because it is able to look at the whole execution sequence and needs only to make one prediction per session. Bauer used this approach to tailor the recognizer to particular users, but it could serve as the starting point for some sort of (semi)automatic labeling. Of course, the success of such an approach depends on the

quality of the recognizer used to produce the labels. A potential problem with this kind of automatic labeling, of course, is that recognition errors in the labeling plan recognizer will be propagated in the corpus.

6.5 Generating Artificial Corpora

In contrast to human data collection, we propose the use of an AI planner and Monte-Carlo simulation to stochastically generate *artificial* plan corpora. This method can potentially be used for any domain and can provide a corpus accurately labeled with goal and hierarchical plan structure. It also provides a cheap way to produce the kind of large corpora needed for machine learning. The general method is as follows:

1. We modify an AI planner to search for valid plans non-deterministically.
2. We model the desired domain for the planner.
3. The algorithm does the following to generate each item in the corpus:
    (a) Stochastically generates a goal.
    (b) Stochastically generates a start state.
    (c) Uses the planner to find a valid plan for generated goal and start state.

We first describe our modifications to an AI planner. Then we discuss issues of domain modeling. We then discuss stochastic generation of the goal and then of the start state. Finally, we discuss the characteristics of corpora generated by this process.

6.5.1 Planner Modification

For plan recognition, we want to create corpora which model all possible plans in the domain a user may have. Typical AI planners do not support this, as most return the

same plan for a given goal and start state. Many planners also try to optimize some plan property (like length or cost) and therefore would seldom output longer, less optimal plans. We want to include all valid plans for a goal in our corpus so that we have broader coverage of the domain.

We, therefore, modified the SHOP2 planner [Nau et al.2003] to randomly generate one of the set of all possible plans for a given goal and start state.[10] We did this by identifying key decisions points in the planner and randomizing the order that they were searched.

SHOP2 [Nau et al.2003] is a sound and complete hierarchical transition network (HTN) planner. SHOP2 is novel in that it searches plan steps in the order they will be executed, which allows it to handle complex reasoning capabilities like axiomatic inference and calls to external programs. It also allows partially ordered subtasks. The planning model in SHOP2 consists of *methods* (decomposable goals), *operators* (atomic actions), and *axioms* (facts about the state).

In searching the state space, there are three types of applicable decisions points, which represent branches in the search space:[11]

- Which (sub)goal to work on next.
- Which method to use for a goal.
- Which value to bind to a parameter.

In order to provide for completeness, SHOP2 keeps lists of all possibilities for a decision point so that it may backtrack if necessary. We modified the planner so that

[10] In principle, the corpus generation technique described here is possible using any planner. The only caveat is that the planner must be randomized, which may or may not be a straightforward thing to do. One of the reasons we chose SHOP2 was its small code base and a modular design that was amenable to randomization.

[11] There is also a fourth which deals with `:immediate` tasks, but that is beyond the scope of this discussion.

these lists are randomized after they are populated but before they are used. This one-time randomization guarantees that we search in a random order but also allows us to preserve the soundness and completeness of the algorithm. We believe our randomized version is equivalent to computing all valid plans and randomly choosing one.

6.5.2 Domain Modeling

Each new domain must be modeled for the planner, just as it would if the intent were to use the planner for its usual purpose. As opposed to modeling for plan generation, however, care should be taken to model the domain such that it can encompass all anticipated user plans.

Usually the planning model must be written by hand, although work has been done on (semi-)automating the process (e.g., [Bauer1998]). Note that, in addition to the model of the plan library, which is also used in many plan recognizers, it is also necessary to model state information for the planner.

6.5.3 Goal Generation

With a randomized planner and a domain model, the corpus generator can generate a random plan sessions given a goal and start state. We stochastically generate both of these. In this section, we discuss both the process of generating goals and the additional domain information that is needed for it. The following session gives a parallel discussion about start states.

We separate goal generation into two steps: generating the goal schema and generating parameter values for the schema.[12]

[12]Note, these steps are very similar to how we stochastically generated goals for users in the Linux corpus, as discussed above.

Goal Schema Generation

In addition to the domain model for the planner, the domain modeler needs to provide a list of possible top-level goals in the domain, together with their a priori probability. A priori probabilities of goals are usually not known, but they could be estimated by the domain modeler's intuitions (or perhaps by a small human corpus). The algorithm uses this list to stochastically pick one of the goal schemas.

Goal Parameter Value Generation

In domains where goals are modeled with parameters, the values of the parameters must also be generated. Goal parameter values can be generated by using one of two techniques. For goal schemas where the parameter values are more or less independent, the domain modeler can give a list of possible parameter values for each slot, along with their a priori probabilities. For schemas where parameter values are not independent, each possible set of parameters is given, along with their probabilities.

Once the goal schema has been chosen, the algorithm uses these lists to stochastically generate values for each parameter in the schema. At this point, a fully-instantiated goal has been generated.

6.5.4 Start State Generation

In addition to a top-level goal, planners also need to know the state of the world — the start state. In order to model agent behavior correctly, we need to stochastically generate start states, as these can have a big effect on the plan an agent chooses.

Generating the start state is not as straightforward as goal generation for several reasons. First, in all but the simplest domains, it will not be feasible to enumerate all possible start states (let alone assign them a priori probabilities). Second, in order to make the planning fast, we need to generate a start state from which the generated goal

is achievable. Practically, most planners (including SHOP2) are **very slow** when given an impossible goal, as they must search through the entire search space before they notice that the goal is impossible.

For these reasons, only a start state which makes the generated goal achievable should be generated. Unfortunately, we know of no general way of doing this.[13] We do believe, however, that some general techniques can be used for start state generation. We discuss these here.

The approach we have chosen is to separate the state model into two parts: fixed and variable. In the *fixed* part, we represent all facts about the state that should be constant across sessions. This includes such things as fixed properties of objects and fixed facts about the state (for example, the existence of certain objects, the location of cities, and so on).

The *variable* part of the state contains facts which should be stochastically generated. Even with the fixed/variable separation, this part will probably not be a set of independent stochastically generated facts. Instead, the domain modeler must come up with code to do this, taking into account, among other things, domain objects, their attributes, and other facts in the state. It is likely that values of sets of facts will need to be fixed simultaneously, especially in cases where they are mutually exclusive, or one implies another, etc. This process will also likely need to be closely linked to the actual goal which has been generated to ensure achievability. In Section 6.6, we describe in more detail how we generated goals and start states for the Monroe corpus.

6.5.5 The Resulting Corpus

A corpus generated by the process described above will contain a complex distribution of plan sessions. This distribution results from the interaction between (a) the a priori probabilities of top-level goals, (b) the probabilities of top-level goal parameter values,

[13] One possibility might be backchaining from the goal state, although we have not explored this.

(c) the algorithm for generating start states, and (d) information encoded in the plan library itself. Thus, although it cannot be used to compute the a priori probabilities of top-level goals and parameter values (which are given as input to the generator), it can be used to e.g., model the probabilities of subgoals and atomic actions in the domain. This is information which cannot be learned directly from the plan library, since the recipes and variable fillers used are also dependent on e.g., the start state.

6.5.6 Related Corpus Generation Work

Conceptually, this method for artificial corpus generation is based on work in NLP which uses grammars to stochastically generate artificial corpora for training language models for speech recognition [Kellner1998]. Of course, there are many differences in methodology. Surface string generation from a stochastic grammar typically assumes no context (state), whereas state is very important in plan recognition. Also, in surface string generation, there is no "goal" which restricts acceptable output.

Probably the closest work to this from the plan recognition field was done by Lesh [1998], who used the Toast reactive planer [Agre and Horswill1992] to generate action sequences given a goal. However, none of the generation process was stochastic. It appears that goals were hand-generated, the state was constant, and the planner was not modified to make decisions non-deterministically, meaning that it always produced the same action sequence given the same set of goals.

6.6 The Monroe Corpus

The Monroe corpus is in an emergency response domain set in Monroe County, New York, based roughly on the domain described in [Stent2000]. We created a plan library with top-level goals such as setting up a temporary shelter and providing medical attention to victims — all top-level goal schemas can be found in Appendix D.

	Linux	**Monroe**
Total Sessions	457	5000
Goal Schemas	19	10
Action Schemas	43	30
Ave Actions/Session	6.1	9.5
Subgoal Schemas	N/A	28
Ave Subgoal Depth	N/A	3.8
Max Subgoal Depth	N/A	9

Table 6.2: Comparison of the Linux and Monroe Corpora

Table 6.2 shows a comparison of the contents of the Monroe corpus and the (post-processed) Linux corpus. The Monroe corpus consists of 5000 plan sessions with an average of 9.5 actions per session. The number of total sessions was, of course, artificially set and could have easily been changed. The 5000 sessions were generated on a high-end desktop computer in under 10 minutes.

In addition to the information we gave earlier about Linux (a goal-labeled corpus), we add several fields here particular to hierarchical corpora. The Monroe corpus has, in addition to the 10 top-level goal schemas, 38 subgoal schemas. The plans in the corpus were on average 3.8 subgoals deep. This measures how many nodes away each atomic action is from the top-level goal. The deepest atomic action in the corpus was 9 levels away from the top-level goal.

In the rest of this section, we discuss the generation of goals and start states in order to illustrate what may be needed in moving to a new domain (in addition to the creation of a plan library).

6.6.1 Goal and Start State Generation

As mentioned above, the plan library includes 10 goal schemas which are specially marked as top-level goals (the difference is not specified in SHOP2 itself). In addition, we added a priori probabilities to each of the goal schemas.

The goal schema was chosen based on those probabilities as discussed above. The schema is then passed to a function which generates the parameter values and the start state simultaneously. In particular, we start with the fixed start state, then stochastically generate locations for movable objects, and then generate other domain facts based on goal schema specific code. We mention these in order here.

Fixed State

The fixed state consists mostly of fixed locations (such as towns and hospitals), objects and their properties. It also includes inference rules supported in SHOP2 which represent things like object types and properties (e.g., adult(x) $\Rightarrow$ can-drive(x)).

Object Locations

As part of the variable state, we define a set of *movable* objects. They are movable in the sense that we can randomly choose where they were located (such as ambulances and workers). We define a list of *sets* of objects, for which it is not important *where* they are located, but only that all objects in the set are in the same location (such as a vehicle and its driver). We also define a list of possible locations, which is used to generate a random location for each object set. (Note, we ensure in the fixed state that locations are fully connected so we do not have to worry about goal impossibility at this step.)

Goal Schema Specific

The rest of the state is created, together with parameter values, in goal schema specific functions. In the emergency domain these were typically very simple, usually just determining which object to use for parameter values.

An example of a more complicated example is that of the goal schema of clearing a road wreck, which takes a wrecked car as a parameter. As we do not model the set of all possible cars in the world, we automatically generate a unique car object as well as its necessary properties (e.g., that it's wrecked, its location). Note that in cases where extra properties are generated, these are also stochastically generated from a priori probabilities (e.g., whether or not the roads are snowy).

6.7 Plan Corpora: Human vs. Artificial

In this section, we raise several issues about the utility of artificial generation of plan corpora versus the collection of human plan corpora. As we have just begun to generate and use such corpora, we do not believe we are in a position to definitively answer these questions. Rather, we raise the questions and give some initial thoughts, which we hope can lead to a discussion in the plan recognition community. The questions treat three general areas: The effort needed to generate artificial corpora; the accuracy of such corpora; and the general power of the technique.

Effort Obviously, the technique we describe above requires a certain amount of work. Minimally, one needs to create a plan library as well as an algorithm for generating start states. Plan library creation is known to be difficult and is a problem for the planning community in general (cf. [Bauer1998]). This may not be a unique problem to artificial corpora, however, as a plan library would likely be necessary anyway in hand-labeling human corpora (at least for plan-labeled corpora). Start state generation is also not trivial, although in our experience, it was much less work than building the plan library.

The main question which needs to be answered here is how the effort to create the machinery for generating an artificial plan corpus compares to the effort needed to gather and annotate a human corpus. Before we can answer this, we not only need more experience in generating artificial corpora, but also experience in producing human corpora — especially plan-labeled corpora.

Accuracy Another point is how accurately an artificial corpus can model human behavior. Ideally, to test this, one would want to gather a human corpus and independently generate an artificial corpus in the same domain and then make some sort of comparison. Of course, care must be taken here, as we suspect that the accuracy of an artificial corpus will be highly-dependent on the plan library as well as the algorithm for generating start states. Another, more practical, evaluation would be the comparison of the performance of a plan recognizer on human data when it has been trained on artificial data versus human data.

Power Another question is in which situations an artificial corpus could be successfully used to approximate human behavior. The technique presented here makes the simplifying assumption (which is also present in most plan recognizers) that an agent first creates an entire plan and then executes it, and that each action is successfully executed. This obviously will not work well in domains where actions fail and replanning is necessary. In future work, we would like to adapt this technique to use an artificial agent, instead of a planner, to plan and simulate execution of the plan in creating a corpus. This would allow us to simulate such phenomena as action failure, replanning, and so forth. In general, we believe that the techniques reported here can build on existing work in agents in modeling human behavior and can be useful in most domains of interest in plan recognition.

6.8 Conclusions and Future Work

There is a shortage of corpora which could be used for plan recognition. However, such corpora could be used both to train probabilistic recognizers as well as to evaluate and compare performance of different recognizers.

We first described a human corpus we created: the Linux corpus. This corpus was gathered from real users and contains action sequences labeled with a top-level goal.

As human data can be expensive and difficult to collect, we also presented a new technique for generating plan-labeled plan corpora using a randomized AI planner and stochastically generated world states. We also presented the Monroe corpus, which was generated using this technique.

In future work, we want to move beyond just plans, and model an actual agent. We believe this would allow us to more closely model agents that we would want to perform plan recognition on, and would include phenomena such as plan failure and replanning. This corpus generation method would allow us to have access to this additional information (when an action failed, when replanning occurs), which would not be readily available from human collection.

7 Flat Goal Recognition

In this chapter,[1] we describe our statistical flat goal recognizer and report its performance on the Linux and Monroe corpora. In Section 7.1, we set up the problem of flat goal recognition mathematically. We split the problem of recognition into two problems: recognition of the goal schema (Section 7.2), and recognition of its parameter values (Section 7.3). We then put the two parts together in Section 7.4 to form an *instantiated* goal recognizer — which recognizes a goal schema along with its parameter values. We give some concluding comments in Section 7.5.

7.1 Problem Formulation

In this section, we set up the problem of flat goal recognition statistically. Before we do that, however, we need to make a few preliminary definitions.

7.1.1 Preliminary Definitions

For a given domain, we define a set of goal schemas, each taking q parameters, and a set of action schemas, each taking r parameters. If actual goal and action schemas do

[1] Some contents of this chapter were reported in [Blaylock and Allen2003; Blaylock and Allen2004; Blaylock and Allen2005c].

not have the same number of parameters as the others, we can easily pad with 'dummy' parameters which always take the same value.[2]

Given an instantiated goal or action, it is convenient to refer to the schema of which it is an instance as well as each of its individual parameter values. We define a function `Schema` that, for any instantiated action or goal, returns the corresponding schema. As a shorthand, we use $X^S \equiv \texttt{Schema}(X)$, where X is an instantiated action or goal.

To refer to parameter values, we define a function `Param` which returns the value of the kth parameter value of an instantiated goal or action. As a shorthand we use $X^k \equiv \texttt{Param}(X, k)$, where X is again an instantiated action or goal.

As another shorthand, we refer to number sequences by their endpoints:

$$1, n \equiv 1, 2, \ldots, n$$

This allows us to shorten definitions in the following ways:

$$A_{1,n} \equiv A_1, A_2, \ldots, A_n$$

$$A_{1,n}^{1,r} \equiv A_1^1, A_1^2, \ldots, A_1^r, A_2^1, A_2^2, \ldots, A_{n-1}^r, A_n^1, \ldots, A_n^r$$

7.1.2 Statistical Goal Recognition

We define flat goal recognition as a classification task: given an observed sequence of n instantiated actions observed thus far ($A_{1,n}$), find the most likely instantiated goal g:

$$g = \text{argmax}_G \, P(G|A_{1,n}) \tag{7.1}$$

[2] The requirement that goal and action schemas have the same number of parameters is for convenience in the mathematical analysis. Below we report how this is circumstance is handled within the recognizer itself.

Using the notation introduced above for referencing schemas and parameter values, we can expand goal and actions into their schema and parameter components:[3]

$$g = \text{argmax}_{G^S, G^{1,q}} \ P(G^S, G^{1,q} | A^S_{1,n}, A^{1,r}_{1,n}) \tag{7.2}$$

Here G^S refers to the goal schema of G and $G^1 \ldots G^q$ refer to G's q parameter values. Each action is similarly decomposed into an action schema and r parameter values. Note that, for now, we assume that each goal has exactly q parameters, and each action has r parameters.

Independence Assumptions

We make two simplifying assumptions at this point, in order to make recognition more tractable. First, we assume that goal parameters are independent of one another, and second, that goal schemas are independent from action parameters (given their action schemas). We now discuss each in more detail.

Goal Parameter Independence We make the simplifying assumption that all goal parameters are independent of one another. This allowed us to separate the probability of each parameter value into independent terms in Equation 7.3. This is, of course, not always the case — an obvious example from the Linux domain is that the source and destination parameters for a copy goal should not have the same value. However, in many cases it appears that they are fairly independent.

Goal Schema and Action Parameter Independence We also assume that a goal schema is independent from an action's parameter values, given the action schema, which allows us to simplify the first term in Equation 7.3. This is also admittedly not always the case. In the Monroe domain, the `call` action describes a telephone call,

[3]From now on we drop the argmax subscript when context makes it obvious.

with one parameter: the recipient of the call. This is used in the domain to turn off power to a particular location or to declare a curfew, as well as other things. The first use always has a power company as a parameter value whereas the second use includes a call to the local police chief.

Although conditioning on parameter values could be informative, it is likely that it would introduce sparsity problems because of the large number of possible parameter values.

Given these two assumptions, Equation 7.2 can be rewritten as:

$$g = \text{argmax } P(G^S|A^S_{1,n}) \prod_{j=1}^{q} P(G^j|G^S, A^S_{1,n}, A^{1,r}_{1,n}) \tag{7.3}$$

Here, the first term describes the probability of the goal schema G^S, which we use for goal schema recognition (Section 7.2). The other terms describe the probability of each individual goal parameter G^j, which we estimate with our goal parameter recognizer (Section 7.3).

7.2 Goal Schema Recognition

We model goal schema recognition on the first term from Equation 7.3 above:

$$g^S = \text{argmax } P(G^S|A^S_{1,n}) \tag{7.4}$$

This gives us a goal schema recognizer, which predicts a top-level goal schema given a list of observed action schemas. In the remainder of this section, we first describe the algorithm used for goal schema recognition and then report the results on test cases from the Monroe and Linux.

7.2.1 Algorithm

Using Bayes' Rule, Equation 7.4 becomes:

$$g^S = \text{argmax} \frac{P(A^S_{1,n}|G^S)P(G^S)}{P(A^S_{1,n})} \tag{7.5}$$

Since $P(A^S_{1,n})$ is constant in the argmax, we can drop it:

$$g^S = \text{argmax } P(A^S_{1,n}|G^S)P(G^S) \tag{7.6}$$

Using the Chain Rule, we can rewrite this as:

$$g^S = \text{argmax} \quad P(A^S_n|A^S_{1,n-1}, G^S)P(A^S_{n-1}|A^S_{1,n-2}, G^S) \ldots P(A^S_1|G^S)P(G^S) \tag{7.7}$$

These conditional distributions are very large and difficult to estimate, therefore, we make an n-gram assumption, i.e., we assume that an action schema A^S_i is only dependent on the goal schema G^S and the j action schemas preceding it ($A^S_{i-j,i-1}$).

For example, if we assume that A^S_i is independent of everything but G^S and A^S_{i-1}, we get a bigram model:

$$g^S = \text{argmax } P(G^S) \prod_{i=2}^{n} P(A^S_i|A^S_{i-1}, G^S) \tag{7.8}$$

We use data from a plan corpus to estimate the a priori goal schema probabilities as well as the n-gram action probabilities.

We have created a goal schema recognizer based on this n-gram model. We describe it here in three phases: First the setup phase, which is run at the start of the recognition session. The update and prediction phases are then run after each action observation.

Setup Phase At the beginning of a recognition session, we create a probability distribution for each of the possible goal schemas in the domain. Each goal schema is assigned its a priori probability ($P(G^S)$), as computed learned from the training corpus.

Update Phase Upon each action observation, we calculate the corresponding bigram probability for each goal schema (for the bigram model $P(A_i^S|A_{i-1}^S, G^S)$). For smoothing, we use an n-gram backoff strategy when the n-gram was not seen in the training data. If an n-gram probability is not found in the model, we use the $n-1$-gram probability multiplied by some discount factor γ. This process is recursive: if the $n-1$-gram probability is not found, then we back off to the $n-2$-gram probability, with an additional penalty factor of γ (the total penalty would now be γ^2). This recurses until a probability is found, or until it goes past the unigram probability. In the latter case, we return a very low probability instead of zero so that we do not ever exclude any goal schema from consideration.

The schema probability distribution is updated calculated by multiplying each goal schema by the corresponding n-gram probability.

Prediction Phase Once the goal schema distribution has been updated, the recognizer now has the option of making a prediction. Unlike other (plan or goal) recognizers of which we are aware, our recognizer supports *selective prediction*, which means that it only makes predictions when it has achieved a certain degree of confidence in the prediction.

We believe this is an important feature for a recognizer. In all but the most trivial domains, it is likely not possible to achieve correct prediction after every observed action, even for humans (cf. [Schmidt, Sridharan, and Goodson1978]). Perfect performance would mean that we would immediately know what the agent was doing after seeing just one action. Things are not always that clear. In the Linux domain, for example, a first action of `pwd`, which has very little predictive power, was not uncommon. Also, for goals like `move-files-by-name` it was not always clear that the goal was a move (as opposed to a copy) until the very last action had been performed (which was then typically the `mv` command).

Instead of having the recognizer make a prediction after each observed action, we

Action		`move-file`	`know-usage`	**Prediction**
(init)	a priori probabilities	**.30**	**.70**	*N/A*
`pwd`	P(G| *init*, `pwd`)	.50	.50	
	new probabilities	**.30**	**.70**	*(no prediction)*
`ls`	P(G| `pwd,ls`)	.30	.70	
	new probabilities	**.16**	**.84**	`know-usage`
`mv`	P(G| `ls,mv`)	.97	.03	
	new probabilities	**.86**	**.14**	**`move-file`**
`ls`	P(G| `mv,ls`)	.80	.20	
	new probabilities	**.96**	**.04**	**`move-file`**

Figure 7.1: Schema Recognition Example: $\tau = 0.8$

set a confidence threshold τ, which allows the recognizer to decide whether or not it is confident enough to make a prediction. If the probability of the prediction is greater than τ, the recognizer predicts. Otherwise, it predicts "don't know."

Another feature supported by the recognizer is n-best prediction. For some applications where the result of goal recognition is used for further reasoning (e.g., natural language understanding), we do not necessarily need a single prediction, but instead can predict the n best goal schemas. In the case of an n-best prediction, the probability of the prediction is taken to be the sum of the probabilities of the n individual goals, and that is then compared against τ in deciding whether to make a prediction.

Example To illustrate the algorithm, we give a short (contrived) example here. In this example domain, there are just two possible goal schemas: `know-usage` (know the disk usage of a particular file) and `move-file` (move a file to a certain directory). Figure 7.1 shows the observed actions and resulting calculations and predictions in an example plan session.

The first line (labeled 'init') shows the a priori probabilities for the goal schemas before any action has been observed. Upon observing the first action (pwd), the recognizer looks up the bigram probabilities of each of the goal schemas (shown by $P(G|$ *init*,pwd$)$). In this case, both of these probabilities is 0.5, as shown in the figure. For each goal schema, the bigram probabilities are then multiplied by the previous prediction probabilities (i.e., the a priori probabilities) and then normalized to a probability distribution, shown on the line labeled 'new probabilities'. The recognize then chooses the goal schema with the highest probability (know-usage) and then compares the probability to the prediction threshold τ. In this case, the probability of the prediction (0.7) is not greater than the threshold (0.8) and thus, no prediction is made.

For the next observed action (ls), the procedure is the same, except this time, the recently calculated prediction probabilities are used instead of the a priori probabilities. Here, the probability of know-usage is greater than the threshold, and thus, this schema is predicted by the recognizer.

The final two predictions occur in much the same way. Bigram probabilities are looked up and new prediction probabilities are calculated. In both cases, the correct schema (move-file) is predicted.

Complexity

As discussed in Chapter 5, prediction speed and scalability with respect to number of goals is a needed feature for goal recognizers. For this reason, we measure complexity in terms of the number of possible goal schemas ($|G|$).

At a prediction opportunity (i.e., after an action has been observed), the update of a single goal schema can be done in constant time (it is a probability lookup with possible n-gram backoff). The entire update phase, then, is linear in the number of goal schemas $O(|G|)$. We do the prediction phase during this pass over the goal schemas as well, keeping track of the n schemas with the highest probabilities. Thus the entire

prediction algorithm is $O(|G|)$, or linear with respect to the number of possible goal schemas.

Comparison with Albrecht et al.

In Chapter 5, we described the goal recognizer developed by Albrecht et al. [1998] and mentioned it is similar to our own flat goal schema recognizer. Now that we have described our schema recognizer, we are in a position to discuss similarities and differences.

At a conceptual level, our recognizer is almost identical to their actionModel recognizer, with the exception that they also condition the probability of the current goal schema on the previous goal schema. (We assume that there is only one goal schema per session.) They, however, also introduce other models which also incorporate the current state in the form of player location and condition the goal schema probability on that as well. They also use their models to predict next player action and next player location, whereas we only predict the goal schema.

Our recognizer uses an $n-1$ backoff strategy for unseen action/goal combinations, whereas they do not. Also, our recognizer uses a threshold to selectively make predictions.

7.2.2 Experiments

We tested our goal schema recognizer on both Monroe and Linux corpora. We first we discuss the general metrics we use for evaluating results of goal schema recognition and then we discuss the experimental results on the two corpora.

Evaluation Metrics

As Lesh [1998] has pointed out, there is a lack of agreed-upon benchmarks and metrics for reporting results in the plan and goal recognition community. This makes it diffi-

cult if not impossible to compare performance across recognizers. As we mention in Chapter 6, one of the contributions of this thesis is a pair of new corpora (Linux and Monroe) which can be used as benchmarks for the community. In this chapter and the next, we also contribute several new evaluation metrics which are designed to measure the desired features of recognizers discussed in Chapter 5.

In reporting results for goal schema recognition, we use the following metrics:

- *Precision:* the number of correct predictions divided by the total number of predictions made.

- *Recall:* the number of correct predictions divided by the total number of actions observed.

- *Convergence:* whether or not the final prediction was correct (i.e., whether the recognizer *finished* the session with the correct answer).

- *Convergence point:* if the recognizer converged, at which point in the input it started giving only the correct answer. This is reported as a quotient of the action number (i.e., after observing x actions) over the total number of actions for that case.[4] This is similar to Lesh's measurement of work saved [Lesh1998].

Precision and *recall* are used to measure overall accuracy of the recognizer, both in predicting and deciding when to predict. It is important to remember that here the predictions are 'online', i.e., that they occur after each observed action, and not post hoc, after all observations have been seen.

Convergence and *convergence point* are an attempt to measure early prediction, i.e., how far into the plan session does the recognizer zero in on the correct prediction. We

[4] It is necessary to report the total number of actions as well. Because this statistic is only for the test cases which converged, it is possible that the average actions per session is different from that of the entire corpus.

Precision	(2/3) 66.7%
Recall	(2/4) 50.0%
Convergence	yes
Convergence Point	3.0/4.0

Figure 7.2: Evaluation Metrics for Example in Figure 7.1

use the term convergence here, as it is often the case that the recognizer is unsure at the start of a session, but that at some point it has seen enough evidence to converge on a particular prediction, which it then begins predicting and predicts from that point on (cf. [Albrecht, Zukerman, and Nicholson1998]). Note that, for the purposes of calculating the *convergence point* if the recognizer does not make a prediction (i.e., predicts "don't know"), it is considered an incorrect prediction and the convergence point is reset, even if the correct prediction was made beforehand.

To illustrate, we refer to the short example given above in Figure 7.1. Each of the above metrics for this example are shown in Figure 7.2. To calculate precision, we see that three predictions were made and two of them were correct, giving us 66.7 percent. For recall, we note there were four possible prediction points (one after each observed action), and again, two correct predictions were made, giving us 50.0 percent.

The example does converge, since the last prediction was correct, therefore we can calculate a convergence point for it. After the third observed action, the recognizer made the right prediction, and it kept making this prediction throughout the rest of the plan session. Thus, we have a convergence point of 3.0 observed actions, over the total of 4.0 observed actions. Note that for a group of results, the denominator will be the average of total observed actions per each converged plan session.

n-best (τ)	**1** (0.7)	**2** (0.9)	**3** (0.9)
Precision	95.6%	99.4%	98.9%
Recall	55.2%	58.7%	69.6%
Convergence	96.4%	99.8%	100.0%
Convergence Point	5.4/10.2	5.4/10.3	4.1/10.2

Table 7.1: Goal Schema Recognition Results on the Monroe Corpus

Monroe Experiments

In our experiments with the Monroe corpus, we randomly selected 500 plan sessions as a test set and trained a bigram model over actions (as discussed above) using the remaining 4500 sessions.

Table 7.1 shows results for different n-best prediction values.[5] We get a precision of 95.6 percent for 1-best prediction, which can be raised to 99.4 percent by predicting the 2 best schemas. In 2-best prediction, the correct schema is eventually predicted for 99.8 percent of sessions. For 1-best, the recognizer converges on the correct schema after seeing an average of 5.4 of 10.2 actions (for those cases which converge). This means that, on average, the recognizer zeros in on the prediction a little more than halfway through the session.

Recall for 1-best is 55.2 percent, which increases to 69.6 percent for 3-best prediction. Although this may seem poor in comparison to precision and convergence numbers in the 90's, it is important to keep in mind that, as we mention above, a recall of 100 percent is usually out of the question, as that would mean that we can always predict the right goal after seeing just one action. We believe 56.2 percent recall (or 70.1 percent for 3-best) to be a very good result.

[5]The threshold value τ needs to be individually set for each n-best value. The τ values here were chosen experimentally.

n-best (τ)	**1** (0.4)	**2** (0.6)	**3** (0.9)
Precision	37.6%	64.1%	73.7%
Recall	22.9%	40.6%	41.4%
Convergence	37.4%	56.5%	59.7%
Convergence Point	3.5/5.9	4.0/7.2	4.1/7.2

Table 7.2: Goal Schema Recognition Results on the Linux Corpus

Linux Experiments

Because of the smaller size of the Linux corpus, we used cross-validation, testing on sets of 5 plan sessions at a time and training a bigram model on the remaining 452.

Table 7.2 shows results for different n-best values. This is a much different picture than in the Monroe domain. Precision in the 1-best case is only 37.6 percent, with 22.9 percent recall and 37.4 percent of sessions converging. Interestingly, the convergence point for 1-best is comparable to that in Monroe.

Although still not near the performance in the Monroe domain, the 2 and 3-best results on the Linux corpus are much better than 1-best, with precision jumping to 64.1 percent for 2-best and 73.7 percent in 3-best, with recall increasing to 40.6 percent and then 41.4 percent. These results are much more reasonable than the 1-best case.

Still, performance on the Linux corpus is much worse than that on the Monroe corpus. We believe there are several contributing factors:

First, the corpora have very different properties. Most prominently, the Monroe corpus has only 10 goal schemas, whereas Linux has almost double that amount (19). In addition, the average session length in Monroe is 9.5 actions whereas in Linux it is only 6.1. Longer sessions can give more evidence to the recognizer, giving more opportunities to make predictions with more evidence. Also, in the Monroe tests, the recognizer was trained on a factor of magnitude more data (4500 sessions) than in the Linux tests (452 sessions).

In addition, as discussed in Chapter 6, the Linux corpus is data from real humans, whereas Monroe is artificially generated. Although we did some automatic cleanup of the Linux corpus (as described in Appendix C), many human “errors” still survived. In one example, a user used the command `grep` to try to search for a file in a directory tree (`grep` is for searching for text in files, not files in a directory). Such mistakes served as a kind of red herring for the recognizer, pointing it strongly in the wrong direction (in this case a text search), and often ruining results for an entire session.

Another factor seems to be goal similarity. Some of the goal schemas used in the Linux corpus are very similar (e.g., `find-file-by-ext` and `find-file-by-name`). The recognizer often confused these (and other similar sets of goals). This is one of the reasons for the big performance increase in the 2 and 3-best prediction results.

7.3 Goal Parameter Recognition

In order to recognize instantiated goals, we need to recognize goal parameter values as well as goal schemas. One straightforward way of doing this would be to treat instantiated goal schemas as atomic goals and then use the goal schema recognition algorithm from above. Thus, instead of estimating $P(\textit{move-files-by-name}|\textit{ls},\textit{cd})$, we would estimate $P(\textit{move-files-by-name}(\textit{a.txt},\textit{bdir})|\textit{ls}(\textit{papers}),\textit{cd}(\textit{progs}))$.

This solution has several problems. First, this would result in an exponential increase in the number of goals, as we would have to consider all possible ground instances. This would seriously impact the speed of the algorithm. It would also affect data sparseness, as the likelihood to have seen any n-gram in the training data will decrease substantially.

For this reason, we perform goal schema and parameter recognition separately, as described in Equation 7.3 above. From the last term of the equation, we get the following for a single parameter g^j:

$$g^j = \text{argmax}\ P(G^j|G^S, A^S_{1,n}, A^{1,r}_{1,n}) \tag{7.9}$$

We could estimate this with an n-gram assumption as we did above. However, there are several problems here as well. First, this would make updates at least linear in the number of objects in the world (the domain of g^j), which may be expensive in domains with many objects. Second, even without a large object space, we may run into data sparsity problems, since we are including both the action schemas and their parameter values. In addition, this model would not work for domains (like Linux) where domain objects (e.g., files) can be created or destroyed during a plan session.

The solutions above also miss out on the generalization that, oftentimes, the *positions* of parameters are more important than their values. For example, the first parameter (i.e., the *source file*) of the action `mv` is usually the `filename` parameter of the goal `move-files-by-name`, whereas the second parameter (i.e., the *destination*) almost never is, regardless of the parameter's actual value.

For our parameter recognizer, we learn probability distributions of equality over goal and action parameter positions. During recognition, we use these distributions along with a special, tractable case of Dempster-Shafer Theory to dynamically create a set of possible parameter values and our confidence of them, which we use to estimate Equation 7.9.

In this section we first describe this model and then report on experiments using it on the Monroe and Linux corpora.

7.3.1 Algorithm

Formally, we want to base our recognizer on the following probability distribution: $P((G^j = A^k_i)|G^S, A^S_i)$, which represents the probability that the value of the kth parameter of action A_i is equal to the jth parameter of the goal G, given both the goal and action schemas as well as the two parameter positions. Note that in this distribution,

the *value* of the parameter is not considered, only its *position*. We can easily compute this conditional probability distribution from our training corpus.

To use the above model to predict the value of each goal schema parameter as we observe actions, we need to be able to combine probabilities for each parameter in the observed action, as well as probabilities from action to action. In order to do this tractably, we have introduce a special subset of Dempster-Shafer Theory (DST) which we call *singleton Dempster-Schafer Theory* (sDST). We first give a short introduction to DST, and then describe sDST. Then we describe the recognition algorithm and its computational complexity.

Dempster-Shafer Theory

Dempster-Shafer Theory (DST)[6] is a generalization of probability theory which allows for incomplete knowledge. Given a domain Ω, a probability mass is assigned to each subset of Ω, as opposed to each element, as in classical probability theory. Such an assignment is called a *basic probability assignment* (bpa).

Assigning a probability mass to a subset in a bpa means that we place that level of confidence in the subset, but cannot be any more specific. For example, suppose we are considering the outcome of a die roll ($\Omega = \{1, 2, 3, 4, 5, 6\}$).[7] If we have no information, we have a bpa of $m(\Omega = 1)$, i.e., all our probability mass is on Ω. This is because, although we have no information, we are 100 percent certain that *one* of the elements in Ω is the right answer; we just cannot be more specific.

Now suppose we are told that the answer is an even number. In this case, our bpa would be $m(\{2, 4, 6\}) = 1$; we have more information, but we still cannot distinguish between the even numbers. A bpa of $m(\{2, 4, 6\}) = 0.5$ and $m(\{1\}) = 0.5$ would intuitively mean that there is a 50 percent chance that the number is even, and a 50

[6]See [Bauer1995] for a good introduction.

[7]This example is taken from [Bauer1995].

percent chance that it is 1. The subsets of Ω that are assigned non-zero probability mass are called the *focal elements* of the bpa.

An often-cited problem of DST is that the number of possible focal elements of Ω is the number of its subsets, or $2^{|\Omega|}$. This can be a problem both for storage and computation time.

Evidence Combination Two bpas m and m' representing different evidence can be combined into a new bpa using Dempster's rule of combination:

$$(m \oplus m')(A) = \frac{\sum_{B \cap B' = A} m(B)m'(B')}{\sum_{B \cap B' \neq \emptyset} m(B)m'(B')} \tag{7.10}$$

The complexity of computing this is $O(l_m l_{m'} |\Omega|)$, where l_m and $l_{m'}$ are the number of focal elements in m and m', respectively. Basically, the algorithm does set intersection (the $|\Omega|$ term) on each combination of focal elements from m and m'.[8] As the number of focal elements of a bpa can be $2^{|\Omega|}$, the worst case complexity of Dempster's rule of combination is $O(exp(|\Omega|))$, where Ω is the set of objects in the domain.

Singleton Dempster-Schafer Theory

In our goal parameter recognizer, we use a special case of Dempster-Schafer Theory which we term *singleton Dempster-Schafer Theory* (sDST), in which we only allow focal points in a bpa to either be singleton sets, or Ω (the full set). sDST has several nice properties:

First, because we only allow singleton sets and Ω as focal elements, a bpa can have maximally $|\Omega| + 1$ elements. Not only are there a decreased number of possible focal elements in sDST bpas, but set intersection for evidence combining becomes simpler as

[8] We only need consider the focal elements here, since non-focal elements have a probability mass 0, which will always make $(m \oplus m')(A) = 0$.

well. As mentioned above, Dempster's rule of combination performs a set intersection of each combination of focal elements from m and m'. With sDST focal terms, set intersection can be done in constant time. We show this by exploring each possible combination of focal element sets a and b from sDST bpas:

1. *Both a and b are singleton sets:* In this case the single values are compared in constant time. If they are the same, the intersection is a copy of a. If they are different, the intersection is $\emptyset$.

2. *One of a and b is a singleton set and the other is Ω:* In representing Ω in the system, we do not need to actually store each individual value in a set. Rather, we can just use a special variable that marks this focal element as being Ω. In the case that one focal element is Ω, the intersection will always be a copy of the other focal element. No inspection of set contents is necessary, thus this can be done in constant time as well.

3. *Both a and b are Ω:* Actually, this is a special case of the preceding case. When both elements are Ω, then the intersection is Ω, which does not require any special inspection of set contents and can also be done in constant time.

These three cases exhaust the possible focal element combinations in combining two sDST bpas. Thus, the complexity of Dempster's rule of combination of two sDST bpas is $O(l_m l_{m'})$ or $O(|\Omega|^2)$ in the worst case.

sDST is also closed under Dempster's rule of combination. The proof actually follows from the three cases intersection we enumerate above. In all three cases, the resulting set is either: the empty set (in which case it is no longer a focal element), a singleton set, or Ω.

Representing the Model with sDST

As stated above, we estimate $P((G^j = A_i^k)|G^S, A_i^S)$ from the corpus. For a given goal schema G^S and the ith action schema A_i^S, we define a *local bpa* $m_{i,k}^j$ for each goal and action parameter positions j and k s.t. $m_{i,k}^j(\{A_i^k\}) = P((G^j = A_i^k)|G^S, A_i^S)$ and $m_{i,k}^j(\Omega) = P((G^j \neq A_i^k)|G^S, A_i^S)$. This local bpa intuitively describes the evidence of a single goal parameter value from looking at just one parameter position in just one observed action. The bpa has two focal elements: $\{A_i^k\}$, which is a singleton set of the actual action parameter value, and Ω. The probability mass of the singleton set describes our confidence that that value[9] is the goal parameter value. The probability mass of Ω expresses our ignorance, as it did in the die roll example above.[10]

In order to smooth the distribution, we always make sure that elements Ω and A_i^k are given a small probability mass. If either one is has probability mass of 1, a very small value is taken from that and given to the other.

There are several things worth noting here. First, if a goal schema has more than one parameter, we keep track of these and make predictions about them separately. Also, we do not need to represent, enumerate or even *know* the elements of Ω. This means that we can handle domains where the set of possible values is very large, or in which values can be created or destroyed. (Both of these are properties of the Linux domain.)

Combining evidence As mentioned above, we maintain a separate *prediction bpa* m^j for each goal parameter position j. Each of these are initialized as $m^j(\Omega) = 1$, which indicates complete ignorance about the parameter values.

[9]Note that this is the actual instantiated value and not just the position. Two instances of the same action schema with different parameter values will create different bpas.

[10]Note here that Ω is the set of *all* possible domain values and still includes A_i^k. The reason for this is that just because we may not have seen much evidence for A_i^k given the action schema doesn't necessarily mean that A_i^k is *not* the goal parameter value. It just means that we don't yet have much evidence that it *is* the value. We actually ran experiments in which Ω did not include any of the values in the singleton focal elements and, while precision went up, recall dropped significantly.

As we observe actions, we combine evidence within a single action and then among single actions. First, within a single action i, we combine each of the local bpas $m^j_{i,k}$ for each parameter position k, which gives us an *action bpa* m^j_i. This describes the evidence the entire action has given us. Then, we combine the evidence from each observed action to give us an overall *prediction bpa* that describes our confidence in goal parameter values given all observed actions so far. We then use this prediction bpa to make (or not make) predictions.

When we observe an action $A_i(p_1, p_2, \ldots, p_r)$ we create local bpas for each action parameter position $m^j_{i,1} \ldots m^j_{i,r}$. The action bpa m^j_i is the combination of all of these: $m^j_i = m^j_{i,1} \oplus m^j_{i,2} \oplus \ldots \oplus m^j_{i,r}$. The prediction bpa is similarly calculated from all of the action bpas from observed actions: $m^j = m^j_1 \oplus m^j_2 \oplus \ldots \oplus m^j_i$. However, we can calculate this incrementally by calculating $m'^j = m^j \oplus m^j_i$ at each action observation. This allows us to only do 1 action-bpa combination per observed action.

It is worth nothing here that only values that we have seen as an action parameter value will be part of the prediction bpa. Thus, the maximum number of focal elements for a bpa m^j will be the total number of unique action parameters seen, plus one (for Ω). As a corollary, this means that our method will not be able to correctly predict a goal parameter unless its value has been seen as an action parameter value in the current plan session. On the other hand, it means that we can predict parameter values that were never seen in the training data, as long as they appear in the observed actions. In the reported results below, we report results of total recall and also ‘recall/feasible’, which restricts recall to the prediction points at which the algorithm *had access to* the right answer. Admittedly, there are cases in which the correct parameter value could be learned directly from the training corpus, without having been seen in the current session, although it is unclear how often this occurs. In future work, we would like to investigate ways of learning both parameter values and positions.

Prediction At some level, we are using the prediction bpa as an estimation of the term $P(G^j|G^S, A^S_{1,n}, A^{1,r}_{1,n})$ from Equation 7.9 above. However, because the bpa contains Ω, it is not a true probability distribution and cannot provide a direct estimation. Instead, we use Ω as a measure of confidence in deciding whether to make a prediction.

To make an n-best prediction, we take the n singleton sets with the highest probability mass and compare their combined mass with that of Ω. If their mass is greater, we make that prediction. If Ω has a greater mass, we are still too ignorant about the parameter value and hence make no prediction.

In order to more finely control prediction, we add a factor to this comparison which we call *ignorance weight* (ψ). In deciding whether or not to make a prediction, Ω is multiplied by ψ before it is compared with the probability of the prediction. Values of ψ greater than 1 will cause the recognizer only to predict when more sure of the prediction, whereas values between 0 and 1 will cause the recognizer to predict more profusely.

Complexity

The other thing to mention is computational complexity of updating the prediction bpa for a single goal parameter G^j. We first describe the complexity of computing the ith action bpa m^j_i, and then the complexity of combining it with the previous prediction bpa m^j.

To compute m^j_i, we combine r 2-focal-element local bpas, one for each action parameter position. If we do a serial combination of the local bpas (i.e., $m^j_i = ((m^j_{i,1} \oplus m^j_{i,2}) \oplus m^j_{i,3}) \oplus \ldots \oplus m^j_{i,r})$, this results in $r-1$ combinations, where the first bpa is an intermediate composite bpa $\hat{m}^j_i$ and the second is always a 2-element local bpa. Each combination (maximally) adds just 1 subset to $\hat{m}^j_i$ (the other subset is Ω which is always shared). The $(k-1)$th combination result $\hat{m}^j_{i,k-1}$ will have maximum length $k+1$. The combination of that with a local bpa is O($2(k+1)$). Thus, the overall complexity of the

combination of the action bpa is $\sum_{k=1}^{r-1} O(2(k+1)) \approx O(r^2)$, where r is the arity of the observed action.

The action bpa m_i^j is then combined with the previous prediction bpa, which has a maximum size of $r(i-1)+1$ (from the number of possible unique action parameter values seen). The combination of the two bpas is $O(ir^2)$, which, together with the complexity of the computation of the action bpa becomes $O(ir^2 + r^2) \approx O(ir^2)$. r is actually constant here (and should be reasonably small), so we get a complexity of $O(i)$. This is done for each of the q goal parameters, but q is also constant, so we still have $O(i)$. This gives us a fast parameter recognition algorithm which is linear in the number of actions observed so far, and is not dependent on the number of objects in the domain.

7.3.2 Experiments

We tested the goal parameter recognizer on the Monroe and Linux corpora as we did the schema recognizer. We discuss each corpus in turn, after briefly mentioning the metrics we use in reporting results.

Evaluation Metrics

In evaluating the parameter recognizer, we use the same metrics we did for the schema recognizer. We also report two new metrics: *recall/feasible* and *convergence/feasible*, which measure how much recall/convergence the recognizer got from what it could *feasibly* get. As mentioned above, the recognizer can only predict values that it has seen as action parameter values within the current session. Thus *recall/feasible* is the number of correct predictions divided by the number of feasible prediction points, i.e., points at which the goal parameter value had already appeared as an action parameter value. *Convergence/feasible* measures the number of correct last predictions in a similar fashion.

n-best (ψ)	**1** (2.0)	**2** (2.0)	**3** (2.0)
Precision	94.3%	97.6%	98.8%
Recall	27.8%	39.2%	40.0%
Recall/Feasible	55.9%	78.9%	80.6%
Convergence	46.9%	76.2%	76.7%
Conv./Feasible	59.1%	96.1%	96.7%
Convergence Point	5.1/10.0	4.8/9.0	4.7/9.0

Table 7.3: Goal Parameter Recognition Results on the Monroe Corpus

Monroe Experiments

We tested the parameter recognizer on the Monroe corpus in the same way we did the schema recognizer, training the probability model on 4500 sessions and testing on the remaining 500. The results of the tests are shown in Table 7.3.

Here the recognition results are quite good, with high precision even in the 1-best case. Recall and convergence, as can be expected, are lower, with only 27.8 percent recall for 1-best. Looking at the measures of recall/feasible and conv./feasible, however, shows that the algorithm is doing quite good for the cases that it can. In only 49.6 percent of parameter prediction points had the parameter value actually appeared as an action parameter value. In fact, in only 79.4 percent of cases did the parameter value appear as an action parameter value at all. Thus the adjusted recall and convergence measures are much higher.

Convergence point performance is also very encouraging, being at less than half-way through the session, even in the 1-best case. In fact, the feasible convergence point (i.e., the average point in the action stream when the parameter value appears) is 3.8/9.1, which means that the recognizer is converging on the right prediction around only one action after the parameter value appears in the action stream.

n-best (ψ)	**1** (2.0)	**2** (2.0)	**3** (2.0)
Precision	90.9%	93.2%	91.4%
Recall	32.1%	35.8%	37.0%
Recall/Feasible	57.1%	63.8%	65.9%
Convergence	54.4%	60.3%	62.1%
Conv./Feasible	66.2%	73.5%	75.6%
Convergence Point	3.5/6.2	3.4/6.2	3.6/6.4

Table 7.4: Goal Parameter Recognition Results on the Linux Corpus

Linux Experiments

The Linux corpus was tested using cross-validation with the identical test set used for schema recognition. The results are shown in Table 7.4 for various n-best values.

Performance for Linux is also quite good, and is only slightly worse than that for Monroe. Precision starts at 90.9 percent and rises to 93.2 percent in the 2-best case. Interestingly, precision goes down in the 3-best case. This is because the recognizer makes more predictions, as it is more sure of the 3-best prediction, but it appears that the third best value tends not to be the right value, and thus it over-predicts.

Recall, convergence, and convergence point are comparable to performance in Monroe. In the Linux corpus, feasible recall is 56.1 percent and feasible convergence is 82.1 percent. The feasible convergence point is 2.8/6.2, thus the recognizer is recognizing parameter values soon after it sees them.

Some errors in Linux are attributable to typographical errors from the user similar to those described above. For example, when a user types `ls flie` instead of `ls file`, this causes the recognizer to consider `flie` as a possible parameter value.

7.4 Instantiated Goal Recognition

We now turn our attention to building an *instantiated* goal recognizer using the schema and parameter recognizers. This brings us back to our original formulation of goal recognition above, particularly to Equation 7.3. We have a goal schema recognizer which estimates the first term, and a goal parameter recognizer which estimates the each of the terms for each parameter position in a goal schema. Mathematically, we simply need to compute the argmax to get the most likely instantiated goal, although, as we will see, this is not so straightforward, especially if we want to support n-best and partial prediction.

The argmax in Equation 7.3 is an optimization problem over several $(q + 1)$ variables $(G^S, G^{1,q})$, where q is the arity of the goal schema. Although this could mean a big search space, it remains tractable in the 1-best case because of an assumption made above: namely, that goal parameter values are independent of one another (given the goal schema). This means that, given a goal schema g^S, the set of individual argmax results for each goal parameter g^j is guaranteed to be the maximum for that goal schema. This now becomes an optimization problem over just two variables: the goal schema and its parameters.

Although computing the argmax works well in theory, there are several problems with using it in practice. First, it only gives us the 1-best prediction. The search space gets larger if we want an n-best prediction. Second is the problem mentioned earlier about goal schema arity. Straight probability comparisons will not work for goals with different arities, as lower-arity goals will tend to be favored.

Partial prediction is also a problem. We want to support partial predictions by allowing the recognizer to predict a (possible empty) subset of the parameter values for a goal schema. This will allow us to make predictions even in cases where the parameter recognizer is unsure about a specific parameter, and capitalizes on the ability of the stand-alone parameter recognizer to not make a prediction in cases where it is not

certain.

In doing partial predictions, however, we encounter a natural tension. On one hand, we want the predictions to be as specific as possible (e.g., predict as many parameter values as possible). On the other hand, we want high precision and recall for predictions.[11] A recognizer which made only full predictions would give us specific predictions (with all parameters predicted), but would likely have low precision/recall. At the other extreme, we could just predict the goal schema which would give us the best chance for high precision/recall, but no parameter information. Yet another dilemma is how to compare two predictions when one has more predicted parameters than the other.

Because of these problems, we have decided to take a slightly different approach to building our instantiated goal recognizer which capitalizes on the prediction ability of the schema and parameter recognizers as they are.

7.4.1 Algorithm

Our instantiated goal recognizer works as follows: at each observed action, we first run the goal schema recognizer. This makes an n-best prediction of schemas (or does not if the confidence threshold is not reached). If no prediction is made by the schema recognizer, the instantiated recognizer also makes no prediction. If the schema recognizer does make a prediction, we use the parameter recognizer to make (or not make) 1-best predictions for each of the parameter positions for each of the n-best schemas. This automatically gives us partial prediction if a prediction is not made for one or more parameter positions in a schema. The combined results then form the n-best instantiated prediction.

Note that this algorithm does not give us true n-best results for the search space. It instead chooses the n-best goal schemas, and then (selectively) predicts parameters

[11] In a way, specificity adds a third dimension to the existing tension between precision and recall.

for them. A true n-best result would include the possibility of having a goal schema twice, with different predictions for parameters. However, as mentioned above, we did not see an obvious way of deciding between, for example, a goal schema with no parameters predicted, and that same goal schema with one parameter predicted. The latter is guaranteed to not have a lower probability, but it is a more specific prediction. Although we do not provide true n-best prediction, we believe our algorithm provides a natural way of deciding between such cases by appealing to the parameter recognizer itself.

Complexity

For an observed action, the recognizer first runs the schema recognizer ($O(|G|)$) and then runs the parameter recognizer ($O(i)$) for each parameter position (q) of each goal schema ($|G|$).[12] This gives us an overall complexity of $O(|G| + |G|iq)$ or $O(|G|iq)$. As q is constant and small, this becomes $O(|G|i)$, which is linear in the number of goal schemas and the number of actions observed so far. Which is exactly what we need for speed and scalability.

7.4.2 Experiments

We tested the instantiated goal recognizer on the Monroe and Linux test sets using the same procedure as outlined above. On average, the recognition time for Monroe was 0.2 seconds per action, and 0.4 seconds per action for Linux with unoptimized Perl code on a high-end desktop PC.

[12]Note that, although we only need run the parameter recognizer on the n-best schemas to get immediate results, we need to run it for all schemas to keep the probability assignments for the other parameters up to date.

n-best (τ/ψ)	**1** (0.7/2.0)	**2** (0.9/2.0)	**3** (0.9/2.0)
Precision	93.1%	95.8%	96.4%
Recall	53.7%	56.6%	67.8%
ParamPctg	20.6%	21.8%	22.3%
Convergence	94.2%	97.4%	98.6%
ConvParamPctg	40.6%	41.1%	48.4%
Convergence Point	5.4/10.0	5.5/10.1	4.4/10.2

Table 7.5: Instantiated Goal Recognition Results for the Monroe Corpus

Evaluation Metrics

We use the same evaluation metrics for the instantiated recognizer as well did the schema recognizer above. In addition, we use two new measures, designed to measure the specificity of prediction. *ParamPctg* reports, for all correct predictions, the percentage of the parameter values for that goal that were predicted. *ConvParamPctg* reports the same for all sessions which converged.

Monroe Experiments

Results on the Monroe test set are shown in Table 7.5. Performance followed schema recognizer performance quite closely, being slightly lower with the addition of parameter recognition.

The specificity measures of ParamPctg and ConvParamPctg were 20.6 percent and 40.6 percent respectively for the 1-best case,[13] meaning that, on average, a correct prediction had just over a fifth of its parameter values predicted, whereas a correct final prediction had under half predicted. These reflect the recall and convergence perfor-

[13]They also remained fairly constant over the n-best values, although this is likely a reflection of the fact that the recognizer only uses the 1-best prediction from the parameter recognizer, regardless of the n-best value for the overall instantiated recognizer.

n-best (τ/ψ)	**1** (0.4/2.0)	**2** (0.6/2.0)	**3** (0.9/2.0)
Precision	36.3%	60.2%	68.8%
Recall	22.1%	38.1%	38.7%
ParamPctg	51.5%	50.0%	51.6%
Convergence	36.1%	53.8%	56.5%
ConvParamPctg	51.8%	49.0%	49.4%
Convergence Point	3.6/5.8	4.0/7.0	4.1/7.0

Table 7.6: Instantiated Goal Recognition Results for the Linux Corpus

mance of the parameter recognizer (30.5 percent and 49.6 percent, respectively for 1-best).

Linux Experiments

Results on the Linux test set are shown in Table 7.6. These also followed the schema recognizer results fairly closely, being lower with the addition of the parameter predictions.

In the Linux corpus, correct predictions had around half of their parameters instantiated, while the Monroe corpus had just over a fifth. Although both corpora had fairly comparable performance in (stand-alone) parameter recognition, it appears that a greater portion of the correctly predicted goals in the Monroe domain happened to be goals for which the parameter recognizer did not have as high of recall.

7.5 Conclusion

In this chapter, we have presented a statistical recognizer of top-level instantiated goals and presented results on the Linux and Monroe corpora. The recognizer is fast and scalable (linear in the number of goal schemas and actions observed so far), and supports

partial prediction. The recognizer does very well on the Monroe domain and decently for 2 and 3-best prediction on the Linux corpus.

In addition, we have presented a set of metrics for evaluating the accuracy, early prediction, and prediction specificity of recognizers. We hope these will be adopted by the community to facilitate easier comparison of goal recognizers.

8 Hierarchical Goal Recognition

In the previous chapter, we introduced an algorithm for flat goal recognition, or recognition of a agent's top-level goal given observed actions. In this chapter, we move to the case of *hierarchical* goal recognition — recognition of the chain of an agent's active subgoals within a hierarchical plan.

Recognizing such chains of active subgoals (henceforth *goal chains*) can provide valuable information not available from a flat recognizer. First, though not a full plan, a goal chain not only provides information about which goal and agent is pursuing, but also a partial description of *how*.

Additionally, the prediction of subgoals can be seen as a type of partial prediction. As mentioned in previous chapters, when a full prediction is not available, a recognizing agent can often make use of partial predictions. In our flat recognizer, we allowed partial prediction through the possible omission of predictions of parameter values. A hierarchical recognizer can additionally predict an agent's subgoals, even when it is still not clear what the top-level goal is. This can allow a recognizer to make predictions much earlier than it can predict top-level goals. We suspect that the longer (in actions) and more involved a plan is, the longer, on average, it will take to recognize the top-level goal. In fact, there is evidence that humans use subgoal prediction as a type of partial prediction [Schmidt, Sridharan, and Goodson1978], especially in interpreting

language [Carberry1990b].

In building our hierarchical goal recognizer, we take the same basic approach we did to flat goal recognition. In Section 8.1, we present a hierarchical goal schema recognizer and in Section 8.2, a hierarchical goal parameter recognizer. We then discuss how the two are combined into a hierarchical instantiated goal recognizer in Section 8.3. We then conclude in Section 8.4.

8.1 Goal Schema Recognition

Hierarchical goal schema recognition can be described as the following problem: Given a set of observed atomic actions ($A_{1,n}$) determine the agent's top-level goal (G) as well as the chain of subgoals ($S_{1,D-1}$) from G to the last observed action (A_n) (where S_1 is the subgoal immediately beneath G and S_{D-1} is the subgoal immediately above A_n). Note that for each subgoal, we indicate its *depth* by how many steps it is away from the top-level goal. We can therefore consider the top-level goal G to actually be S_0, although we will frequently refer to it as G.

As an example, consider the plan tree shown in Figure 8.1. After observing action $A_{1:5}$, the goal chain to be recognized would be ($G : S_{1:1} : S_{2:3}$), the (sub)goal nodes which lead from the top-level goal to the latest observed action. After observing the next action (A_6), we would want to recognize ($G : S_{1:2} : S_{2:4}$).

In moving to hierarchical recognition, it was our hope to be able to be able to reuse our flat schema recognizer, recognizing immediate subgoals at each level and then using those results to predict subgoals at the next level up. This was unfortunately not possible. The flat schema recognizer is basically a classifier — given an ordered list of observed actions, it labels the plan session with a top-level goal. However, consider the bottom level of the plan tree from Figure 8.1:

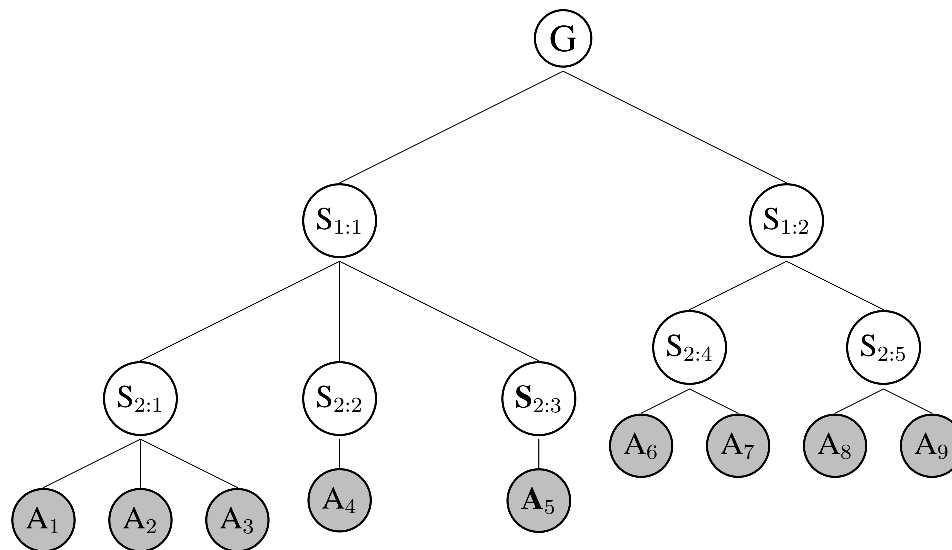
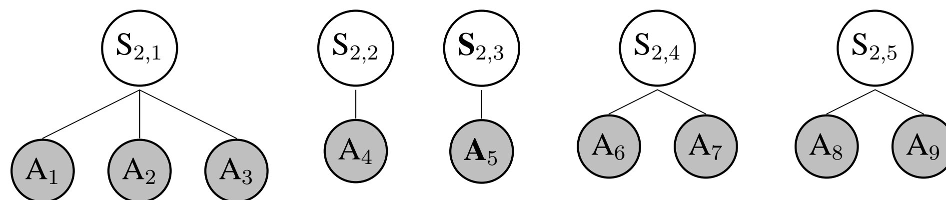
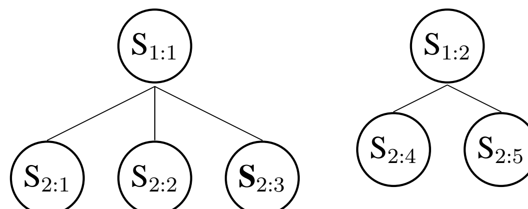


Figure 8.1: An Example Plan Tree



At this level, instead of having a single goal to predict, as we did in flat recognition, we have five. Still worse, we do not know in general how many subgoals we have at the next level, or at which point one ends and the next begins.

When we consider the next level up, things only get worse. Consider recognition of subgoals at level 1:



If we are cascading recognition results up, we would now be treating level 2 as input (i.e., observed actions). At the atomic action level (level D), we know that there is a

new action at each timestep. At this middle level, we do not know when each subgoal at level 2 (S_2) ends, as well as not knowing when each subgoal at level 1 (S_1) ends. In addition, we are also uncertain of our observed output (S_2), whereas the flat recognizer assumed that we observed A with certainty.

As we discussed in Chapter 5, other hierarchical goal recognizers (e.g., [Pynadath and Wellman2000; Bui, Venkatesh, and West2002]) deal with these problems by taking an approach similar to parsing. In both approaches, the equivalent of productions are used to define legal sequences of nodes at the level below each subgoal. We have chosen to take a different approach, based on the forward algorithm in Hidden Markov Models, which allows us to perform recognition without the need of specified production rules.

In the remainder of this section, we first discuss a new type of graphical model used in our recognition algorithm and how we use it to represent plans. We then describe the schema recognition algorithm itself and then report experimental results using the recognizer.

8.1.1 Cascading Hidden Markov Models

In our hierarchical schema recognizer, we utilize a type of graphical model we have termed a *Cascading Hidden Markov Model (CHMM)*, which consists of D stacked state-emission HMMs ($H_{0,D-1}$). Each HMM[1] (H_d) is defined by a 5-tuple ($\sigma_d, \kappa_d, \Pi_d, A_d, B_d$) where σ_d is the set of possible hidden states; κ_d is the set of possible output states; $\Pi_d = \{\pi_{d:i}\}, i \in \sigma_d$ is the initial state probability distribution; $A_d = \{a_{d:ij}\}, i, j \in \sigma_d$ is the set of state transition probabilities; and $B_d = \{b_{d:ik}\}, i \in \sigma_d, k \in \kappa_d$ is the set of output probabilities.

The HMMs are stacked such that for each HMM (H_d), the output state is the hidden state of the HMM below it (H_{d+1}). For the lowest level (H_{D-1}), the output state is the

[1] We use here a similar notation to that in [Jurafsky and Martin2000], although they define an arc-emission HMM.

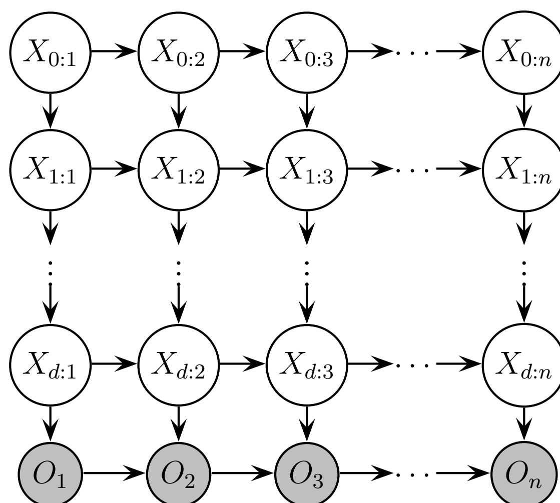


Figure 8.2: A Cascading Hidden Markov Model (CHMM)

actual observed output. In essence, at each timestep t, we have a chain of hidden state variables ($X_{0,D-1:t}$) connected to a single observed output O_t at the bottom level. An example of a CHMM is shown in Figure 8.2.

Here, the dth HMM (i.e., the HMM which starts with the hidden state $X_{d:1}$) is a normal HMM with the output sequence $O_{1,n}$. As we go up a CHMM, the hidden level becomes the output level for the level above it, and so forth.

We will now discuss the differences between CHMMs and other hierarchical HMMs. We then discuss how inference is done with CHMMs, in particular, how the forward probability is calculated, as this is a key part of our recognition algorithm.

Comparison to Hierarchical HMMs

Hierarchical HMMs (HHMMs) [Fine, Singer, and Tishby1998] and the closely related Abstract HMMs (AHMMs) [Bui, Venkatesh, and West2002] represent hierarchical information using a limited-depth stack of HMMs. In these models, an hidden state can output either a single observation, or a string of observations. Each observation can also be associated with a hidden state at the next level down, which can also output observations, and so forth. When a hidden state outputs an observation, control is transferred

to that observation, which can also output and pass control. Control is only returned to the upper-level when the output observation has finished its output. This is similar in function to a push-down automaton although it is not equivalent, as HHMMs only support a finite depth.

In contrast, a CHMM is much simpler. Here, each hidden state can only output a single observation, thus keeping the HMMs at each level in lock-step. In other words, in CHMMs, each level transitions at each timestep, whereas only a subset transitions in HHMMs.

Below, we use CHMMs to represent an agent's execution of a hierarchical plan. As we will discuss there, mapping a hierarchical plan onto a CHMM results in a loss of information which could be retained by using an HHMM (cf. [Bui, Venkatesh, and West2002]). However, using CHMMs allows us to do tractable online inference in terms of the number of possible states (subgoals). Exact reasoning in HHMMs has been shown to be exponential in the number of possible states [Murphy and Paskin2001].

Computing the Forward Probability in CHMMs

An analysis of the various kinds of inference possible with CHMMs is beyond the scope of this thesis. Here we only focus on the forward algorithm, which is used in our schema recognition algorithm below.

Normal HMMs In an HMM, the forward probability

$$\alpha_i(t) = P(o_{1,t}, X_t = i | \Pi, A, B)$$

describes the probability of the sequence of outputs observed up until time t ($o_{1,t}$) and that the current state X_t is i, given an HMM model (Π, A, B).

The set of forward probabilities for a given timestep T, ($\alpha(T) = \{\alpha_i(T), i \in \sigma\}$) can be efficiently computed using the so-called forward algorithm. The forward algo-

rithm uses a state lattice (over time) to compute the forward probability of all intermediate states. This allows it to efficiently compute forward probabilities for the next timestep by simply using those from the previous timestep, using dynamic programming. The algorithm works as follows:

First, $\alpha(0)$ is initialized with the initial state probabilities (Π). Then, for each subsequent timestep t, individual forward probabilities are computed using the following formula:

$$\alpha_j(t) = \left[\sum_{i \in \sigma} \alpha_i(t-1) a_{ij} \right] b_{jo_t} \tag{8.1}$$

The complexity of computing the forward probabilities for a sequence of T observations is O($|\sigma|^2 T$) (where σ is the set of possible hidden states). However, as we will be using the forward probability in making online predictions in the next section, we are more interested in the complexity for *extending* the forward probabilities to a new timestep (i.e., calculating $\alpha(t+1)$ given $\alpha(t)$). For extending to a new timestep, the runtime complexity is O($|\sigma|^2$), or quadratic in the number of possible hidden states.

Algorithm Overview In a CHMM, we want to calculate the forward probabilities for each depth within a given timestep: $\alpha(t) = \{\alpha_d(t)\}, d \in 0, D-1$, where $\alpha_d(t) = \{\alpha_{d:i}(t)\}, i \in \sigma_d$. This can be done a timestep at a time, cascading results up from the lowest level ($D-1$). The basic form of the algorithm is shown in Figure 8.3.

Initialization of each level occurs as normal — as if it were a normal HMM — using the start state probabilities in Π_d. For each observation new o_t, the new forward probabilities for the chain are computed a bottom-up fashion, starting with $\alpha_{D-1}(t)$. At this level, the new forward probabilities can be computed as for a normal HMM using the formula in Equation 8.1.

We then move up the chain, computing one forward probability set at a time, using the results of the lower chain as observed output. However, we cannot use Equation 8.1

1: $t = 0$
2: initialize each $\alpha_d(0)$ as usual (using Π_d)
3: **loop**
4: $t = t + 1$
5: $o_t =$ new observation
6: calculate $\alpha_{D-1}(t)$ given $\alpha_{D-1}(t-1)$ and using o_t as observed output
7: **for** $d = D - 2$ downto 0 **do**
8: calculate $\alpha_d(t)$ given $\alpha_d(t-1)$ and using $\alpha_{d+1}(t)$ as observed output
9: **end for**
10: **end loop**

Figure 8.3: Algorithm for Calculating Forward Algorithm for CHMMs

to calculate these forward probability sets ($\alpha_d(t)$). This is because the normal forward algorithm assumes that output is observed with certainty. While this was the case for level $D - 1$ (where the output variable is o_t), for all other levels, the output state is actually also a hidden state ($X_{d+1:t}$), and is thus uncertain.

We overcome this by first observing that, although we do not know the value of $X_{d+1:t}$ with certainty, if we have the forward probability set for that node ($\alpha_{d+1}(t)$), we can use it as a probability distribution over possible values for the state. As discussed above, we can compute the forward probability set at the bottom level $\alpha_{D-1}(t)$, which gives us a probability distribution over possible values of $X_{D-1:t}$. In order to calculate the forward probability at the next level up $\alpha_{D-2}(t)$ (as well as higher levels), we need to augment the forward algorithm to work for HMMs with uncertain output, which we discuss now.

Computing the Forward Probability with Uncertain Observation The forward algorithm for a single HMM can easily be adjusted to handle the case where the output state is uncertain (i.e., we have a probability distribution over possible values). The initialization step remains the same. We calculate forward probabilities for subsequent timesteps using the following equation (instead of Equation 8.1):

$$\alpha_{d:j}(t) = \left[\sum_{i \in \sigma_d} \alpha_{d:i}(t-1) a_{ij}\right] \left[\sum_{k \in \sigma_{d+1}} \alpha_{d+1:k}(t) b_{jk}\right] \tag{8.2}$$

As in Equation 8.1, the forward probability is calculated as the product of two terms, corresponding to the probability of transition to that state and the probability of the output from the new state. In calculating with uncertain observation, the state transition term remains the same (the sum of all weighted transition probabilities). We change the output probability term to be a weighted sum over all possible outputs (k) multiplied by their output probabilities (b_{jk}). This sum gives us the total probability that whatever was output was output when the HMM was in state j.

Algorithm Complexity The complexity of computing the forward probability with uncertain output at a level d for T timesteps is $O(T(|\sigma_d|^2 + |\sigma_d||\sigma_{d+1}|))$ where the term $|\sigma_d||\sigma_{d+1}|$ comes from the summing over output probabilities. If we assume that the set of possible states is roughly the same at each level ($|\sigma_{d+1}| \approx |\sigma_d|$), the complexity becomes $O(T|\sigma_d|^2)$, which is unchanged from the complexity of the forward algorithm with certain output.

As mentioned above, because we are making online predictions, we are also interested in the complexity of *extending* forward probabilities to the next timestep. In this case, this also remains the same as that for the normal forward algorithm, and is $O(|\sigma_d|^2)$, or quadratic in the number of possible states at each level.

In a CHMM, calculating the next chain of forward probabilities, as described in Figure 8.3, simply calculates the next forward probabilities for each level. Thus the overall complexity of extending the chain given a new observation is $O(D|\sigma_{max}|^2)$, where σ_{max} is the level with the most possible states, and D is the depth of the CHMM.

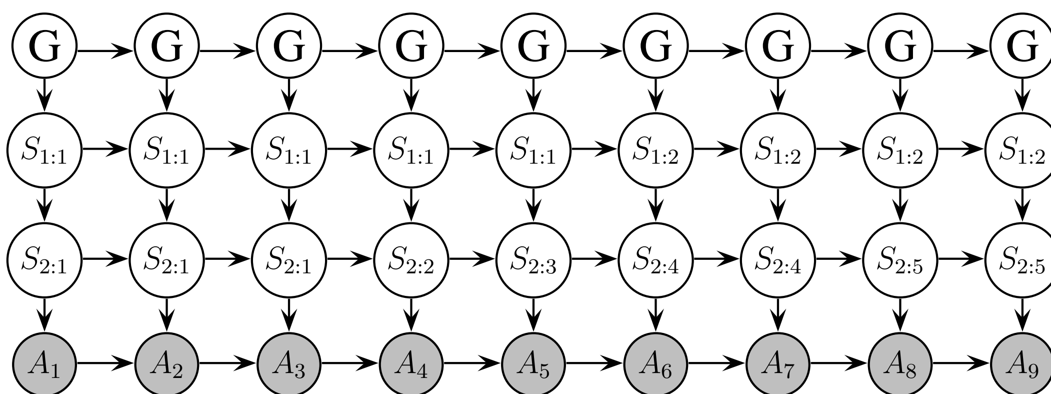


Figure 8.4: The Sequence of Goal Chains Corresponding to the Plan Tree in Figure 8.1

8.1.2 Mapping Plan Trees onto CHMMs

Our basic approach to hierarchical goal schema recognition is to model a plan tree as a CHMM and to use forward probabilities to make predictions at each subgoal level. Given a plan-labeled corpus, we convert it to a sequence of goal chains, which we can then use to learn transition and output probabilities for the schema recognizer.

Up until now, we have modeled plans as trees (e.g., as shown in Figure 8.1). In hierarchical goal recognition, however, we do not try to recognize the entire tree (which would be *plan* recognition), but rather the *goal chain*, or sequence of subgoals from the last observed action to the top-level goal. We can, in this way, convert a plan tree into a sequence of these goal chains, one for each observed atomic action. For example, the results of converting the plan tree in Figure 8.1 into a list of goal chains is shown in Figure 8.4.

Note that subgoals which span more than one timestep are simply duplicated across all timesteps in that span. Below, we will discuss the ramifications of this expansion, including the fact that, at upper levels, sequence information is lost. First, however, we must discuss one more issue that must be dealt with. The goal chains in the plan tree in Figure 8.1 are all the same depth. This, however, may not always be the case. A CHMM, however, is required to be of uniform depth. We now discuss how this case is

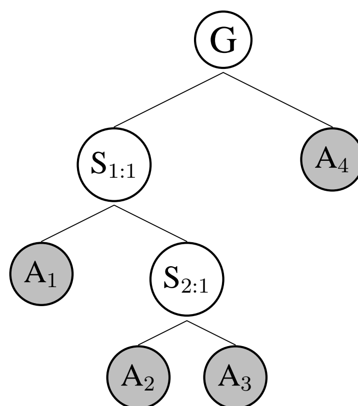


Figure 8.5: A Plan Tree with Varying Depth

handled.

Handling Differences in Plan Tree Depth

Paths in a plan tree need not necessarily be of the same depth. We model plan decomposition through recipes which can include subgoals as well as atomic actions, at any level. As an example, consider the plan tree in Figure 8.5. Here, observed actions ($A_{1,4}$) exist at various depths in the tree. Actions A_2 and A_3 are the deepest at depth 3, whereas A_1 is at depth 2 (it is the child of subgoal $S_{1:1}$). Note the case of A_4, which is a direct child of the top-level goal G.

As mentioned above, in order to convert trees to CHMMs, we need to have leaves all be at the same depth. To do this, we expand each leaf which is too shallow by copying the node which is its immediate parent and inserting it between the parent and the leaf. We repeat this until the leaf node is at the proper depth. We refer to these copies of subgoals as *ghost* nodes. The result of expanding the tree in Figure 8.5 is shown in Figure 8.6, which can then be converted into the CHMM shown in Figure 8.7. Note that by doing this expansion, we make each possible subgoal at a particular depth S_d a possible subgoal at each subsequent depth, as it can be copied to lower levels as a ghost. As a simplification below, we will just assume that each subgoal is possible at

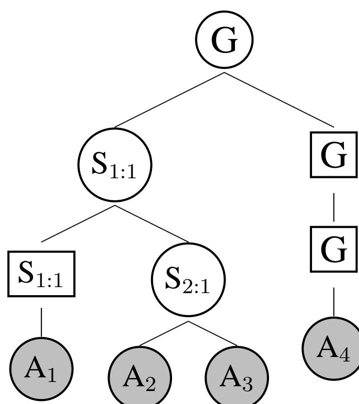


Figure 8.6: The Expanded Version of the Plan Tree in Figure 8.5: (ghost nodes are shown as rectangles.)

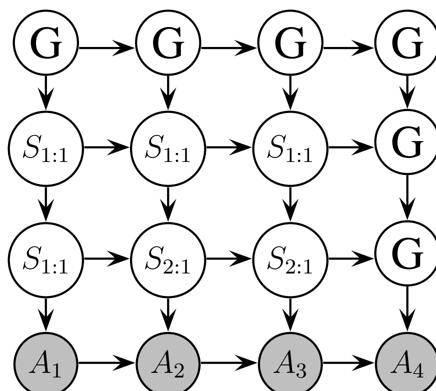


Figure 8.7: The Sequence of Goal Chains Corresponding to the Expanded Plan Tree in Figure 8.6

each depth. Thus, we will simply use S to refer to the set of all possible subgoals (at any level, including top level goals).

Discussion

In mapping plan trees to CHMMs, we lose certain information about tree structure, which is retained in approaches which do goal schema recognition using a tree-like

structure (e.g., Abstract HMMs [Bui, Venkatesh, and West2002] or grammars [Pynadath and Wellman2000]). Consider again the plan tree from Figure 8.1 along with its corresponding CHMM (in Figure 8.4). At depth 1 (i.e., the first level below the top-level goal), two subgoals are executed: $S_{1:1}$ and $S_{1:2}$. This transition occurs after timestep 5, and the information that the preceding subgoal was $S_{1:1}$ can be used in making predictions at timestep 6. In subsequent timesteps, however, we loose this information because of the Markovian assumption. Thus, at timestep 7, the HMM at level 1 thinks that the previous subgoal was $S_{1:2}$, although the last *actual* subgoal was $S_{1:1}$. Furthermore, for cases where a subgoal of a certain type is followed by a subgoal of the same type, it becomes impossible to determine if these comprise one or two instances of the subgoal in the original tree.

As we discussed briefly above, the advantage to making this simplification is improved runtime complexity. Exact inference in hierarchical HMMs has been shown to be exponential in the number of possible states [Murphy and Paskin2001], while we have shown that computing forward probabilities in CHMMs is only quadratic in the number of possible states.

8.1.3 Recognition Algorithm

Schema recognition is performed by constructing a CHMM and using the forward probabilities to make predictions at each subgoal depth. We first describe how the CHMM is trained, and then how predictions are made. We then analyze the runtime complexity of the recognition algorithm.

Training the CHMM

As a CHMM is really just a stack of HMMs, we need to learn transition probabilities (A_d), output probabilities (B_d) and start state probabilities (Π_d) for each depth d. These

are estimated from a plan-labeled corpus in which each session is converted into a sequence of goal chains, as described above.

Predictions

At the start of a recognition session, a CHMM of appropriate depth for the domain is initialized with start state probabilities from the model. Upon observing a new action, we calculate the new forward probabilities for each depth using the CHMM forward algorithm described in Figure 8.3.

Using the forward probabilities, n-best predictions are made separately for each depth, using the same prediction method used in our flat schema recognizer (as described in Section 7.2). The n most likely schemas are chosen, and their combined probability is compared against a confidence threshold τ.[2] If the n-best probability is greater than τ, a prediction is made. Otherwise, the recognizer does not predict at that level for that timestep.

It is important to note that using this prediction algorithm means that it is possible that, for a given timestep, subgoal schemas may not be predicted at all depths. It is even possible (and actually occurs in our experiments described below) that the depths at which predictions occur can be discontinuous, e.g., a prediction could occur at levels 4, 3, and 1, but not 2 or 0. We believe this to be a valuable feature as subgoals at different levels may be more certain than levels below, depending on the domain.

Complexity

The runtime complexity of the recognizer for each new observed timestep is the same as that of forward probability extension in the CHMM: $O(D|S|^2)$, where D is depth of the deepest possible goal chain in the domain (not including the observed action), and

[2] Although it would be possible to set a separate threshold for each depth, our results below are based on using a single threshold for all levels.

S is the set of possible subgoals (at any level). Thus the algorithm is linear in the depth of the domain and quadratic in the number of possible subgoals in the domain.

8.1.4 Experiments

We now report on two sets of experiments using the hierarchical goal schema recognizer. For both experiments, we used the Monroe corpus, divided into the same set of training and testing data as used for the experiments on flat recognition described in Chapter 7. Note that we did not perform experiments on the Linux corpus, as our algorithm requires a plan-labeled corpus for learning the CHMM, and the Linux corpus is only top-level goal labeled.

Before we describe the experiments and their results, however, we describe how we report results.

Result Reporting

We report results for individual subgoal depths, as well as totals. For each depth, we use the same metrics we used for flat schema recognition in Chapter 7: *precision*, *recall*, *convergence*, and *convergence point*. However, as there are some differences with flat recognition, we make two changes to how these are counted.

First, as described above, in modeling goal recognition, when a plan tree has leaf nodes at differing depths, we insert *ghost nodes* above the shallow leaves. When we are performing goal recognition, the assumption is that each goal chain is the same length, and predictions are potentially made at each level. Upon observing a new action, it is unknown at which depth of the plan tree it was before extending occurred — the recognizer simply makes predictions for the entire goal chain, possibly including ghost nodes.

In reporting results for each level (and in the total), we *do not* count predictions when the correct answer was a ghost node. Ghost node prediction tends to be correct,

and thus resulted in heavily inflated results, especially at lower depths. The introduction of ghost nodes is a product of our use of CHMMs, and thus it is unfair to credit these as correct predictions.

The second change we make to result reporting involves convergence and convergence point. Subgoals may only correspond to one timestep (e.g., they only result in one executed atomic action), in which case, it does not make sense to report convergence or a convergence point. For all levels, we only report convergence and convergence point for subgoals which correspond to at least 2 timesteps.

The Pure Forward Algorithm

We first tested the algorithm as described above on the same training and testing data from the Monroe corpus as used in the experiments in Chapter 7. The results of the test are shown in Table 8.1. We first look at the results for predicting top-level goal schemas (level 0) and then explore the other levels.

Top-level Results In interpreting the results, we first refer back to the results of flat schema recognition reported in Chapter 7, on the same data set. For convenience, we show the results of flat recognition again in Table 8.2.

Of course, flat recognition was only concerned with predicting the top-level goal, which is the same as level 0 for the hierarchical recognizer. For recall, convergence, and convergence point, the two recognizers perform fairly equivalently, both in 1-best and 2-best prediction. Precision, however, is markedly lower in the hierarchical recognizer, for both the 1-best and 2-best cases. Whereas precision is 95.6 percent for 1-best in the flat recognizer, it drops to 85.7 percent for the hierarchical recognizer. A similar drop in precision from 99.4 percent to 91.5 percent is shown in the 2-best case.

Although there seem to be several factors involved in this drop, it is perhaps most important to mention two. First is, as we mention above, the loss of true bigram information within the hierarchical recognizer. In the hierarchical recognizer, the top-level

	1-best ($\tau = 0.7$)				2-best ($\tau = 0.95$)			
level	**prec.**	**recall**	**conv.**	**conv. pt**	**prec.**	**recall**	**conv.**	**conv. pt**
0	85.7%	55.7%	97.0%	5.3/10.2	91.5%	57.4%	97.0%	5.1/10.2
1	87.1%	42.4%	61.2%	3.7/6.5	99.7%	51.4%	72.1%	3.2/6.1
2	69.1%	35.3%	45.3%	3.5/4.8	100%	24.3%	45.7%	4.1/4.8
3	70.2%	32.6%	30.1%	2.1/3.1	95.9%	79.2%	86.8%	3.2/4.5
4	66.0%	54.7%	61.8%	3.3/3.7	92.3%	79.1%	87.3%	2.4/3.7
5	59.0%	45.9%	6.2%	3.8/4.2	98.8%	98.8%	100%	1.2/3.9
6	69.3%	69.3%	0.0%	N/A	100%	100%	100%	1.0/4.0
7	95.2%	95.2%	N/A	N/A	100%	100%	N/A	N/A
8	100%	100%	N/A	N/A	100%	100%	N/A	N/A
Total	76.6%	45.1%	59.4%	4.1/7.0	95.7%	57.8%	74.4%	3.7/6.5

Table 8.1: Results of Schema Recognition using the CHMM

	1-best ($\tau = 0.7$)				2-best ($\tau = 0.95$)			
level	**prec.**	**recall**	**conv.**	**conv. pt**	**prec.**	**recall**	**conv.**	**conv. pt**
top	95.6%	55.2%	96.4%	5.4/10.2	99.4%	58.7%	99.8%	5.4/10.3

Table 8.2: Results of Flat Schema Recognition on the Monroe Corpus (from Chapter 7)

goal is predicted based on predictions at the next immediate subgoal level (level 1) as opposed to directly from the action observation level as is the flat recognizer. As mentioned above, converting a plan tree into a sequence of goal chains can loose explicit information about the actual previous subgoal.

Secondly, and most importantly, a direct comparison of algorithm performance is difficult because the hierarchical is doing much more than simple top-level goal classification as was done in the flat recognizer. As we discuss at the beginning of this chapter, direct application of the flat algorithm was not possible for hierarchical goal

recognition. The hierarchical recognizer presented here not only recognizes goals at the top level, but at every subgoal level as well. The top-level goal is a special case because there was only one per session. Arguably, we could improve performance by using the hierarchical recognizer for the subgoal levels and then the flat recognizer for top-level recognition, although this then looses the generalization that the hierarchical recognizer can also handle cases where several top-level goals are pursued serially.

Other Levels Results at lower levels are not as good as those at the top level. The foremost reason is that there is actually more competition at lower levels. At lower levels, many more subgoals are possible (even top-level goals, through ghost extending), whereas only the 10 top-level schemas are possible at level 0. Also, there are several lower-level subgoals per level throughout a goal session. Only one top-level goal makes the transition probabilities much simpler at the top level as well (basically transition probabilities are 1 between the same schemas and 0 between any others).

That said, in the 1-best case, recognition results are fairly good for levels 1 and 7, and 8, although there is a trough between them. A partial explanation is that, at higher levels, there are less competitors (because of higher-level subgoals can appear as ghosts at lower levels, but not vice versa). Thus, as we move to lower levels, things become harder to predict. At the same time, the lower we go, the closer to the observed output, and thus closer to certain information. Thus the last two levels have very good precision and recall because they are so closely related to the observed action. (Levels 7 and 8 contained no subgoals which span more than one timestep, hence convergence and convergence point are not reported.)

It appears that in the middle (e.g., levels 2-6), the recognizer tends to not be able to distinguish well among competitors. That this is the case can be shown by looking at the 2-best case, where all levels move to the 90's or 100 percent for precision and also improve dramatically in recall.[3] Thus, for the middle levels, the next best com-

[3] Except for level 2 recall, which seems to be a quirk in the data. This irregularity disappears in our

petitor seems to often be the right one. However, as information is cascaded up the CHMM, middle levels only have the level immediately below them as context for updating probabilities. In a followup set of experiments, we tried to make the recognizer more predictive by adding more information.

Adding Observation Information

In order to try to improve performance, we added observation-level information to the calculations at each level. We did this by making both transition and output probabilities context dependent on the current and last observed action (bigram). The idea was that this would tie upper-level predictions to possible signals present only in the actual actions executed (as opposed to just some higher-level, generic subgoal). This is often done in probabilistic parsing [Charniak1997], where lexical items are included in production probabilities to provide better context.

The only change we made was to the transition probabilities (A_d) and output probabilities (B_d) at each level. Thus, instead of the transition probability $a_{d:ij}$ being $P(X_{d:t} = j | X_{d:t-1} = i)$, we expand it to be conditioned on the observed actions as well:

$$a_{d:ij} = P(X_{d:t} = j | X_{d:t-1} = i, O_t, O_{t-1})$$

Similarly, we added bigram information to the output probabilities ($b_{d:ik}$):

$$b_{d:ik} = P(X_{d:t} = i | X_{d+1:t} = k, O_t, O_{t-1})$$

These distributions were learned from the corpus. We also used the corpus to estimate the corresponding unigram (over observed action) distributions as well as the original transition and output distributions from the last experiments. In the case that a bigram context with the transition or output context had not been seen in the corpus,

---

next experiment.

	1-best ($\tau = 0.7$)				2-best ($\tau = 0.95$)			
level	**prec.**	**recall**	**conv.**	**conv. pt**	**prec.**	**recall**	**conv.**	**conv. pt**
0	85.6%	58.6%	100%	5.2/10.2	90.7%	62.0%	100%	4.9/10.2
1	84.3%	54.8%	71.8%	2.9/6.1	96.1%	77.3%	99.0%	2.3/5.6
2	89.3%	46.3%	45.8%	3.4/4.7	93.0%	64.3%	84.4%	3.5/4.8
3	74.8%	42.8%	41.2%	2.7/3.5	97.6%	80.1%	99.0%	3.5/4.5
4	78.7%	53.5%	61.8%	3.3/3.7	97.0%	73.2%	100%	3.2/3.8
5	59.3%	46.1%	6.2%	3.8/4.2	99.1%	77.1%	100%	2.0/3.9
6	69.3%	69.3%	0.0%	N/A	100%	100%	100%	1.0/4.0
7	95.2%	95.2%	N/A	N/A	100%	100%	N/A	N/A
8	100%	100%	N/A	N/A	100%	100%	N/A	N/A
Total	81.9%	52.3%	65.0%	3.8/6.8	94.9%	71.4%	95.7%	3.3/6.1

Table 8.3: Results of Schema Recognition using the CHMM and Observation Information

we used the unigram and then original distributions as backoff probabilities. At each backoff step, we multiplied the resulting probability by a penalty of 0.5.

The results of using this modified recognizer are shown in Table 8.3.

The addition of observation context resulted in a slight drop in top-level precision, although it did result in an increase in recall.

The real improvement, however, can be seen in the middle levels, where both precision and recall went up in most cases. For example, precision at level 4 rose from 66.0 percent to 78.7 percent and recall at level 2 went from 35.5 percent to 46.3 percent. That there was an overall improvement in both the 1-best an 2-best cases can be seen in comparing the level totals. For 1-best, total precision rose from 76.6 percent to 81.9 percent, and recall rose from 45.1 percent to 52.3 percent.

8.2 Goal Parameter Recognition

In this section, we describe a hierarchical goal parameter recognizer which (selectively) predicts parameter values for each depth in a goal chain. For stand-alone parameter recognition, we make the same assumption we did in Chapter 7, namely, that we know a priori the goal schemas in the chain. We also make the assumption that we know when each subgoal begins and ends. We remove these assumptions when we move to full instantiated recognition in the next section. We also assume a CHMM model of plan execution, where each subgoal level transitions at each timestep.

We first describe the recognition algorithm, and then the results of tests on the Monroe corpus.

8.2.1 Recognition Algorithm

Parameter recognition is performed separately at each depth, with the same basic algorithm used for flat recognition. For each level d, we define a *prediction bpas* $m^{d:j}$, $j \in 1, q$ for each subgoal parameter position j.[4] These prediction bpas are then updated and used to make predictions after each new observed action. We first discuss the initialization phase of the algorithm, then how updates occur at the bottom level. We then discuss updates at upper levels and finally, we analyze the algorithm's complexity.

Initialization

At each depth d, we initialize a set of q prediction bpas $m_0^{d:j}$, $j \in 1, q$ s.t. $m_0^{d:j}(\Omega) = 1$. As parameter recognition is necessarily tied to the corresponding goal schema, each set of prediction bpas is associated with the beginning goal schema at each level $S_{d:1}$. This is similar to what was done in the flat parameter recognizer.

[4] For clarity of discussion, we assume that all goal schemas (and action schemas) have the same number of parameter positions q. In the algorithm itself, this is dealt with in a similar way to that in Chapter 7.

As we will describe below, each of these prediction bpas are recomputed at each timestep and are used to make predictions. As opposed to the flat recognizer, however, in all but the top level, subgoal instances may change during the session. When a new observation corresponds to the start of a new subgoal at a certain level, we reinitialize the prediction bpas at that level before integrating the new evidence. The reasons for this are twofold. First, as mentioned above, parameter recognizers are specific to a certain goal schema. Thus, a parameter recognizer for subgoal schema X cannot be used to recognize parameters for subgoal schema Y. Also, even if the two subgoals instances have the same schema, they will likely have different parameter values. Thus keeping the prediction bpas from the previous recognizer would possibly cause the recognizer to keep predicting the *old* parameter values. What we want to do in either case is start out with a blank slate. We discuss below how to deal with this problem in instantiated recognition, where subgoal changes are not known with certainty.

Updates at the Bottom Level

At the bottom level ($D-1$), we are dealing with certain output (i.e., the observed action) and thus can perform parameter recognition as we did for the flat case. Upon observing action A_t, we calculate a set of local bpas $m_{t,k}^{D-1:j}$, which represent the evidence that the kth action parameter provides for the jth goal parameter. This local bpa is calculated by using the probability that the action parameter value is the same as the goal parameter value given the context, i.e., the following probability: $P((S_{D-1}^{j} = A_t^k)|S_{D-1}^{S}, A_t^S)$.

For the observed action, each of the q local bpas (one for each action parameter position) are combined (using Dempster's rule of combination) to create an action bpa $m_t^{D-1:j}$, which holds the evidence for goal parameter j from action A_t. The prediction bpa $m^{D-1:j}$ is then updated through combination with this action bpa. This is unchanged from the flat recognition algorithm.

Updates at Upper Levels

At higher levels, we need to modify the recognition algorithm because of two complications: uncertain output and multiple output instances which belong to the same event. We discuss each in turn.

Dealing with Uncertain Output As we move up the goal chain, the subgoal schema at level $d+1$ becomes the output action at level d. The parameter recognizer expects an instantiated action as input, and thus we integrate the parameter probabilities from the recognizer at level $d+1$ to form a goal schema with uncertain parameter values. Instead of integrating just the predicted n-best parameter values for each position, we include each prediction bpa for each parameter position for the schema from the level below: ($m^{d+1:j}$).

To handle uncertain parameter values, we change the way each local bpa $m^{d:j}_{t,k}$ is calculated. We first initialize the local bpa to be a copy of the parameter prediction bpa from the level below $m_{d+1:j}$. This is then weighted by the positional equality probability used above: $P((S^j_{D-1} = A^k_t)|S^S_{D-1}, A^S_t)$. Bpa weighting is done using Wu's weighting formula [Wu2003]:

$$m'(A) = \begin{cases} wm(A) & : \quad \text{for all } A : A \subset \Omega, \text{ and } A \neq \Omega \\ wm(A) + 1 - w & : \quad A = \Omega \end{cases} \tag{8.3}$$

where m is the bpa to be weighted and w is the weight. This equation basically weights each of the focal points of the bpa and redistributes lost probability to Ω.

The resulting weighted bpa is then used as the local bpa in further processing.

Dealing with Multiple Output Instances A more subtle change in the algorithm at upper levels arises from the fact that subgoals at the level below may correspond to more than one timestep. As an example, consider again the goal chain sequence shown

in Figure 8.7. At level 2, the subgoal $S_{2:1}$ lasts for 2 timesteps. At the lowest level, we are assured that each observed action is a separate instance of an action.

This becomes a problem because Dempster's rule of combination makes the assumption that combined evidence bpas come from independent events. For the case in Figure 8.7, when predicting parameters for the level 1 subgoal, we would combine output evidence from $S_{2:1}$ two separate times (as two separate action bpas), as if two separate events had occurred.

The parameter distributions for $S_{2:1}$ will of course likely be different at each of the timesteps, reflecting the progression of the parameter recognizer at that level. However, instead of being two independent events, they actually reflect two estimates of the same event, with the last estimate presumably being the most accurate (because it itself has considered more evidence at the output level).

Thus, we need to change the update algorithm to additionally keep track of the prediction bpa formed with evidence from the last timestep of the most recently *ended* subgoal at the level below, which we will call the *last subgoal prediction (lsp) bpa*. At a new timestep, the prediction bpa is formed by combining the action bpa with this lsp bpa. If this timestep does not end the subgoal at the level below, then this prediction bpa is only used to make predictions at this timestep and then is discarded. If the subgoal below does end, then we save this prediction bpa as the new lsp bpa. In this way, we treat evidence from continuing subgoals as updates, instead of new events.

Prediction

Prediction is done separately at each level in the same way it was in the flat recognizer. The n-best parameter values for a given position are chosen, and their combined weight is compared against the ignorance measure Ω multiplied by the specified ignorance weight ψ.

Complexity

To calculate a new prediction bpa for a given parameter position at a given depth, we combine q local bpas (one for each output parameter position) to make an action bpa. This action bpa is then combined with the lsp bpa. This results in q total combinations.

As discussed in Chapter 7, the complexity of combination of 2 sDST bpas is the product of their sizes. In the flat recognizer, local bpas were guaranteed to only have 2 elements, however, this is not the case in hierarchical recognition. Local bpas at upper levels will have the number of elements as the prediction bpa at the level below. As an upper bound for local bpa size, we note that, at timestep t, local bpas can only contain parameter values which have been seen in the observed actions up to that point A_1, t. Thus the maximum number of unique parameter values seen is tq, where q is the maximum arity of observed actions. Thus the complexity of these q combinations is $O(t^2q^3)$.

For a single timestep, we compute q new prediction bpas at D levels, giving us an overall complexity of $O(Dt^2q^4)$. As q is constant (and likely small), we drop the term, making the complexity $O(Dt^2)$ or quadratic in the number of actions observed thus far.

8.2.2 Experimental Results

We tested the recognizer on the Monroe corpus in the same way as the flat recognizer in Chapter 7. The results of the tests are shown in Table 8.4. Results are given using the same metrics used for the flat recognizer and using the same per-level counting scheme used for reporting results for the hierarchical schema recognizer above. We first look at the results at the top level (i.e., level 0) and then the other levels.

Top-level Results

To help interpret the results, we compare performance at the top level to that of the flat recognizer (which only made predictions at the top level). For convenience, the results

	1-best $(\psi = 2.0)$					
level	**prec.**	**recall**	**recall/feas.**	**conv.**	**conv./feas.**	**conv. pt**
0	98.6%	25.8%	52.0%	44.7%	56.3%	5.0/9.9
1	99.7%	26.4%	52.0%	39.9%	55.2%	4.1/6.3
2	96.7%	53.0%	76.4%	51.6%	57.7%	2.5/4.8
3	98.7%	73.8%	89.4%	73.8%	74.1%	3.1/4.1
4	99.3%	80.0%	94.6%	80.9%	80.9%	3.3/3.8
5	97.5%	82.6%	91.1%	53.1%	53.1%	2.2/3.9
6	99.9%	98.3%	99.3%	50.0%	50.0%	2.0/4.0
7	100%	100%	100%	N/A	N/A	N/A
8	100%	100%	100%	N/A	N/A	N/A
total	98.5%	51.7%	76.5%	51.6%	61.2%	3.5/5.7

	2-best $(\psi = 2.0)$					
level	**prec.**	**recall**	**recall/feas.**	**conv.**	**conv./feas.**	**conv. pt**
0	97.7%	40.1%	80.8%	76.0%	95.8%	4.7/9.0
1	99.9%	41.3%	81.2%	63.6%	88.0%	3.5/5.7
2	99.6%	65.9%	95.1%	82.9%	92.8%	2.8/4.7
3	99.8%	81.0%	98.2%	97.6%	97.9%	3.4/4.5
4	100%	83.3%	98.5%	97.6%	97.6%	3.3/3.9
5	100%	89.7%	99.0%	93.0%	93.0%	2.5/3.9
6	100%	99.1%	100%	100%	100%	2.5/4.0
7	100%	100%	100%	N/A	N/A	N/A
8	100%	100%	100%	N/A	N/A	N/A
total	99.5%	62.4%	92.4%	78.6%	93.2%	3.5/5.6

Table 8.4: Results of Parameter Recognition

	1-best ($\psi = 2.0$)					
level	**prec.**	**recall**	**recall/feas.**	**conv.**	**conv./feas.**	**conv. pt**
top	94.3%	27.8%	55.9%	46.9%	59.1%	5.1/10.0

	2-best ($\psi = 2.0$)					
level	**prec.**	**recall**	**recall/feas.**	**conv.**	**conv./feas.**	**conv. pt**
top	97.6%	39.2%	78.9%	76.2%	96.1%	4.8/9.0

Table 8.5: Results of Flat Parameter Recognition on the Monroe Corpus (from Chapter 7)

of the flat parameter recognizer on the same data set are shown in Table 8.5.

The hierarchical recognizer performed slightly better in both the 1-best and 2-best cases. In 1-best, precision moved from 94.3 percent to 98.6 percent, although there was a drop in recall from 27.8 percent to 25.8 percent. In the 2-best recognizer, results were slightly better all around.

The reason for the improvement in performance is likely attributable to the fact that (perfect) subgoal schema information was present in the hierarchical recognizer. This allowed parameter values to be considered given the immediate child subgoal, giving better context for predictions.

Other Levels

The hierarchical recognizer performed well at other levels as well, with precision staying (for the 1-best case) in the high 90's and even up to 100 percent for levels 7 and 8. This performance inched up for the 2-best case (with 100 percent precision for levels 4–8).

It is interesting to note that recall begins quite low (25.8 percent for level 0) and then climbs as we go down levels, reaching 100 percent for levels 7 and 8. As mentioned in

Chapter 7, high absolute recall is not to be expected in plan recognition, as ambiguity is almost always present. The closer we move to the actual observed action, however, the higher precision gets. This can be attributed to two factors. First, subgoals at lower levels are closer to the observed input, and thus deal with less uncertainty about what the parameter values are.

Second, and probably most important, is that lower-level subgoals span fewer timesteps than those at higher levels, meaning that, if parameter values are available, they will be seen after a shorter number of actions. In the case of levels 7 and 8, all subgoals only spanned one timestep, and thus only had one chance to get the right parameter values. It turns out that parameter values at these levels always directly corresponded to the action parameters, which is why precision and recall reach 100 percent here.

Overall, the performance of the parameter recognizer was very encouraging, especially the performance at lower levels which had high recall. This is an important factor in our ability to do specific and accurate partial prediction in the instantiated goal recognizer, which we move to now.

8.3 Instantiated Goal Recognition

In this section, we describe how we integrate the schema and parameter recognizers to create a hierarchical instantiated goal recognizer, which can recognize a chain of subgoal schemas and their parameter values. We first describe the recognition algorithm and then test results on the Monroe corpus.

8.3.1 Recognition Algorithm

The recognition algorithm for the hierarchical recognizer is similar to that of the flat recognizer. Upon observing a new action, we first update the schema recognizer and use it to (selectively) make preliminary predictions. For each of the predicted subgoals

(at each level), we then use the corresponding parameter recognizers to (selectively) make predictions for each of the parameter positions.

We discuss the stages of initialization, update, and prediction, and then present an analysis of the runtime complexity of the algorithm.

Initialization

We initialize the schema recognizer as described above. For each depth, we also initialize a parameter recognizer for each possible subgoal schema. Note that this is different from the stand-alone parameter recognition done above, which assumed a knowledge of subgoals (and their beginning and ending times) and thus had for each level only one active parameter recognizer at a time. Here we will basically be updating $|S|$ parameter recognizers per level per timestep. We describe how updates are handled in the next section. Also, unlike the stand-alone parameter recognizer, these parameter recognizers will run for the entire session. As we do not know when subgoals begin and end, we do not initialize new recognizers during the session.

Update

Given a new observed action, we first update the schema recognizer and use it to make preliminary predictions (as will be discussed in the next section). We then update each of the parameter recognizers. (Note that, as was the case for flat recognition, we need to update each parameter recognizer at each timestep, even if it is not used to make a prediction at that timestep.)

However, we need to modify the parameter recognizer update algorithm to make it work for instantiated recognition. We make three changes which correspond to each of the following issues: uncertain output schemas, uncertain transitions at the prediction level, and uncertain transitions at the output level. We discuss each in turn.

Uncertain Output Schemas In the stand-alone parameter recognizer, we made the assumption that the goal schema was known. At higher levels, this meant that output consisted of a goal schema and uncertain parameter values. At higher levels in the instantiated recognizer, however, we additionally have uncertain goal schemas as output. In a nutshell, we now need to model output as a set of uncertain goal schemas, each having a set of uncertain parameters.

Modifying the update algorithm for this case follows the same principle we used in handling uncertain parameters. To handle uncertain goal schemas, we compute an action bpa for each possible goal schema as described for the stand-alone recognizer. We then introduce a new intermediate result called an *observation bpa* which represents the evidence for a parameter position given an entire observation (i.e., a set of uncertain goal schemas each associated with uncertain parameter values). To compute the observation bpa, first each action bpa in the observation is weighted according to the probability of its goal schema (using Equation 8.3). The observation bpa is then computed as the combination of all of the action bpas. This effectively weights the contributed evidence of each uncertain goal schema according to its probability (as computed by the schema recognizer).

Uncertain Transitions at the Prediction Level In the stand-alone parameter recognizer, we knew a priori when goal schemas at the prediction level began and ended. This information was used to reset prediction bpas to ignore evidence gathered from observed actions corresponding to previous subgoals. To reset the prediction bpas, they were set to $\Omega = 1$, or total ignorance.

In instantiated recognition, we do not know the when goal schemas begin or end. We can, however, provide a rough estimation by using the transition probabilities estimated for the schema recognizer. We use this probability (i.e., the probability that a new schema does not begin at this timestep) to weight the last subgoal prediction (lsp) bpa at each new timestep.

Basically, this provides a type of decay function for evidence gathered from previous timesteps. Assuming we could perfectly predict schema start times, if a new schema started, we would have a 0 probability, and thus weighting would result in a totally ignorant lsp bpa. On the other hand, if a new subgoal did not start, then we would have a weight of 1 and thus use the evidence as it stands.

Uncertain Transitions at the Output Level Not knowing schema start and end times gives us a similar problem at the output level. As we discussed for the stand-alone parameter recognizer, we need a way of distinguishing which observed output represents a new event versus which represents an updated view of the same event.

We handle this case in a similar way to that above. We calculate the probability that the new observation starts a new timestep by the weighted sum of all same transition probabilities at the level below. This estimate is then used to weight the prediction bpa from the last timestep and then combine it with the lsp bpa to form a new lsp bpa. In cases that there is high probability that a new subgoal was begun, the prediction bpa will have a large contribution to the lsp bpa, whereas it will not if the probability is low.

Prediction

Prediction is performed as it was for the flat recognizer. First, the goal schema recognizer is used to (selectively) make an n-best prediction of goal schemas for each depth. If the schema recognizer does not make a prediction at a certain depth, the instantiated recognizer also does not predict for that depth.

If the schema recognizer does make a prediction, we use the corresponding parameter recognizers to predict parameter values for each of the n-best goal schemas. The instantiated prediction then consists of the chain of predicted goal schemas with the instantiated parameters for those values for which the parameter recognizers made a prediction.

Complexity

First, we must analyze the complexity of the modified parameter recognizer (which deals with output with uncertain goal schemas). The modified algorithm computes the observation bpa by combining (worst case) $|S|$ action bpas — each with a maximum size of iq (limited by the number of unique parameter values seen, as described above). Thus, the total complexity for the update of a single parameter position is $O(|S|t^2q^3)$ and for the parameter recognizer of a single goal schema (with q parameter positions), this becomes $O(|S|t^2q^4)$. Again, we drop q as it is constant and small, which gives us $O(|S|t^2)$.

The complexity of an update for the instantiated recognizer can be calculated from the runtime of the schema recognizer plus the runtime of each of the $D|S|$ parameter recognizers (one per each goal schema per level). Thus the total runtime complexity is $O(D|S|^2 + D|S|^2t^2) = O(D|S|^2t^2)$, or quadratic in the number of possible goal schemas and the number of actions observed so far.

8.3.2 Experimental Results

We tested the recognizer on the Monroe corpus in the same way done for the flat recognizer in Chapter 7. The results of the tests are shown in Table 8.6. Results are given using the same metrics used for the flat recognizer and using the per-level counting scheme used for the other hierarchical recognizers. We first look at the results at the top level (i.e., level 0) and then the other levels.

Top-level Results

To help interpret the results, we compare performance at the top level to that of the flat recognizer (which only made predictions at the top level). For convenience, the results of the flat instantiated recognizer on the same data set are shown in Table 8.7.

	1-best $(\tau = 0.7, \psi = 2.0)$					
level	**prec.**	**recall**	**param%**	**conv.**	**conv. param%**	**conv. pt**
0	82.5%	56.4%	24.0%	90.8%	49.8%	5.6/10.3
1	81.3%	52.8%	23.5%	67.6%	26.5%	3.1/6.1
2	85.4%	44.3%	22.5%	45.8%	38.5%	3.4/4.7
3	72.9%	41.7%	82.4%	41.2%	90.6%	3.0/3.5
4	73.6%	50.0%	99.9%	61.8%	100%	3.7/3.7
5	58.8%	45.7%	100%	6.2%	100%	4.2/4.2
6	69.3%	69.3%	100%	0.0%	N/A	N/A
7	95.2%	95.2%	100%	N/A	N/A	N/A
8	100%	100%	100%	N/A	N/A	N/A
total	79.0%	50.4%	44.1%	61.7%	46.4%	3.9/6.8

	2-best $(\tau = 0.95, \psi = 2.0)$					
level	**prec.**	**recall**	**param%**	**conv.**	**conv. param%**	**conv. pt**
0	88.2%	60.2%	23.2%	91.0%	49.9%	5.2/10.3
1	93.8%	75.4%	16.6%	94.8%	18.9%	2.4/5.6
2	89.7%	62.0%	42.1%	84.4%	45.2%	3.6/4.8
3	90.6%	74.4%	81.8%	99.0%	71.0%	3.9/4.5
4	90.8%	68.6%	96.5%	100%	80.9%	3.8/3.8
5	98.2%	76.4%	81.4%	100%	53.1%	2.0/3.9
6	98.3%	98.3%	99.2%	100%	50.0%	4.0/4.0
7	100%	100%	100%	N/A	N/A	N/A
8	100%	100%	100%	N/A	N/A	N/A
total	91.3%	68.7%	47.2%	92.5%	43.7%	3.6/6.1

Table 8.6: Results of Instantiated Recognition

	1-best ($\tau = 0.7, \psi = 2.0$)					
level	**prec.**	**recall**	**param%**	**conv.**	**conv. param%**	**conv. pt**
top	93.1%	53.7%	20.6%	94.2%	40.6%	5.4/10.0

	2-best ($\tau = 0.9, \psi = 2.0$)					
level	**prec.**	**recall**	**param%**	**conv.**	**conv. param%**	**conv. pt**
top	95.8%	56.6%	21.8%	97.4%	41.1%	5.5/10.1

Table 8.7: Results of Flat Instantiated Recognition on the Monroe Corpus (from Chapter 7)

Hierarchical instantiated results at the top level closely mirror results of the hierarchical schema recognizer. This also happened for the flat recognizer and is to be expected, as schema recognition performance limits performance of the instantiated recognizers.

As discussed in Chapter 7, the addition of parameter predictions serves to degrade the precision and recall of schema recognition results. Interestingly, a comparison of the degradation in the flat recognizer (Tables 8.2 and 8.7) and the hierarchical recognizer (Tables 8.3 and 8.6) shows a similar percentage point drop between the schema and instantiated recognizers for precision and recall. The flat schema recognizer achieved 95.6 percent precision and 55.2 percent recall for the 1-best case, which dropped to 93.1 percent and 53.7 percent for the flat instantiated recognizer. Similarly, the hierarchical schema recognizer achieved 85.6 percent precision and 58.6 percent recall for the 1-best case, which dropped to 82.5 percent and 56.4 percent for the hierarchical instantiated recognizer.

The percentage of parameter values instantiated for correct predictions actually increased in the hierarchical recognizer — from 20.6 percent to 24.0 percent, which at least partially reflects the improved performance of the hierarchical parameter recog-

nizer over the flat recognizer. Thus, at the top level, almost a quarter of parameter values are instantiated in correct predictions, which rises to almost half for converged sessions.

8.3.3 Other Levels

Precision and recall at other levels also closely mirror the performance of the schema recognizer. Precision dips in the middle levels as it did in the schema recognizer, but this levels out for 2-best prediction, which achieves precision ranging from the high 80's to 100 percent (with recall ranging in the 60's and 70's for high levels and high 90's and 100 percent for the lower levels).

Parameter prediction for levels 1 and 2 remains in the 20's, with a sudden jump to 82.4 percent at level 3, 99.9 percent at level 4, and 100 percent for the lower levels, for the 1-best level. Note that the drop in parameter prediction at several levels in the 2-best case is due to the fact that the recognizer gets more cases right (i.e., increases recall), but that many of the new correct predictions have less instantiated parameter values. Thus the decrease in number reflects that the recognizer is getting more correct predictions, but it does not reflect a decrease in performance for the cases it got correct in 1-best prediction.

8.4 Conclusion

In this chapter, we have presented a hierarchical goal recognizer which recognizes the chain of active subgoal schemas, instantiated with their parameter values. For efficient hierarchical goal schema recognition, we have introduced a new type of graphical model, the Cascading Hidden Markov Model (CHMM) and use a modified forward algorithm to make predictions based on probabilities learned from a plan-labeled corpus.

We are now in a position to evaluate the hierarchical goal recognizer based on the desired requirements we outlined in Chapter 5:

Speed: This refers to the speed of the algorithm in making a prediction, given a new observation. Our recognizer has a runtime complexity which is quadratic in the number of possible subgoals and the number of observed actions, and linear in the depth of the goal chain. This makes it scalable in terms of all of these factors.

Early Prediction: Not only should a recognizer be fast, it should also be able to predict the agent's goal before the agent completes it. In the Monroe domain, for cases where it converges, our recognizer is on average able to predict the correct top-level goal after a little more than half of the observed actions.

Partial Prediction: In cases where full early prediction is not possible, recognizers should be able to provide partial predictions. Our recognizer provides partial prediction in two separate ways. First, it can make partial predictions by predicting only a subset of parameter values for a goal schema. The early prediction results above are actually based on partially instantiated predictions, and not full predictions.

Also, our recognizer can provide partial information by predicting the agent's current subgoals, even in cases where it is not yet able to predict the top-level goal. This is perhaps the most valuable contribution of the hierarchical recognizer, as it is able to make predictions at lower levels very early on (after just the first action for levels 7 and 8 — with 100 percent precision). As discussed above, as goal complexity increases, it is unlikely that the top-level goal will be predictable early on in the session. In such cases, the ability to predict lower-level subgoals should be even more valuable in allowing the recognizer to make predictions early on in the exchange.

9 Conclusion

The goal of building a generalized agent-based dialogue system is one which requires a lot of progress in many areas of artificial intelligence. In this thesis, we have presented work which lays several foundational pieces for agent-based dialogue systems.

First, we have created a model of agent collaborative problem solving which is based on human communication. This model also includes a descriptive language of communicative intentions which can serve as an (artificial) agent communication language, or as a model of communicative intentions in human dialogue. We have also described a model of dialogue based on this collaborative problem-solving model and expanded to incorporate a well-known theory of communicative grounding. This model of dialogue is able to represent a wider range of dialogue phenomena than previous systems, including a range of collaborative paradigms and collaborative problem-solving activity.

The collaborative problem-solving model and the dialogue model based upon it are an important backbone of agent-based dialogue research. As we discussed in Chapter 1, there are three main subsystems necessary to support agent-based dialogue. First, an interpretation subsystem is necessary to convert language into communicative intentions. The dialogue model provides a descriptive language of these communicative intentions which need to be recognized.

Second, a behavior subsystem is necessary to guide the actions of the system. This is where the autonomous agent lives. The dialogue model presented in this thesis represents dialogue moves and dialogue state at a problem-solving level, which is much closer to a form which current artificial agents reason with than most dialogue models; it provides a significant narrowing of the gap between the two fields of research.

Third, a generation subsystem is required to convert communicative intentions into language for communication. Again, the definition of what these communicative intention was a prerequisite for this.

After the agent-based dialogue model, we turned our attention in the second half of the thesis to supporting the problem of interpretation, more specifically, to intention recognition, where communicative intentions are recognized from a high-level semantic form. Intention recognition is a special form of plan recognition (the recognition of an agent's goal and plan given observations), and one of the biggest challenges to this has been the lack of tractable algorithms.

In the second half of the thesis, we introduced a fast goal recognizer based on statistical machine learning. The algorithm is fast and scalable, with runtime complexity quadratic in the number of possible goals. At the same time, it is able to hierarchically recognize active goal schemas and their parameter values, and does not place the restrictions on the expressiveness of the domain that other, scalable recognizers do. We intend to use this goal recognizer as the engine for an intention recognizer for agent-based dialogue in future work.

In order to train and test the recognizer, we also provided two new corpora to the plan recognition community — the Linux corpus and the Monroe corpus — and introduced a method for the stochastic generation of plan-labeled corpora. We also described a set of general desirable properties of plan recognizers, and introduced several new metrics for measuring these. We believe the contribution of these corpora and metrics will foster better evaluation in the plan recognition community and better comparability between different recognizers.

In short, the work described in this thesis has been a foundational one. We have set the primary foundation for work in agent-based dialogue systems by describing a model of agent-based dialogue and its accompanying communicative intentions. We have also contributed a new form of scalable goal recognition to the plan recognition community, which will serve as the foundation of efficient intention recognition algorithms for agent-based dialogue systems.

In the remainder of this chapter, we discuss various routes of needed future work which this thesis has lead to. We first discuss future work in dialogue modeling, using the dialogue model we presented in Chapters 3 and 4. We then explore future work in the area of goal recognition. Finally, we discuss in more detail, needed work in agent-based dialogue systems.

9.1 Future Work in Dialogue Modeling

In this section, we mention several areas of future work for the collaborative problem-solving dialogue model.

9.1.1 Evaluations and Argumentation

As noted in Chapters 3 and 4, we do not yet have a good idea about how to represent evaluations in the model, beyond a simple good/bad dichotomy. More study needs to be done to determine how evaluations are made in dialogue, and what kinds are distinguished.

As a further extention, we believe it may be possible to use evaluations to model argumentation in dialogue. Typically, argumentation is considered to be part of an exchange in which beliefs of the various agents are supported and attacked — a sort of debate.

Usually, argumentation is modeled solely as being about beliefs (cf. [Chu-Carroll and Carberry2000]). Our evaluations seem to also serve this function. For example, an evaluation of a constraint on the top-level situation (e.g., a belief in our model) could be used to decide whether that belief is good or bad, i.e., true or not true. This is also true for anywhere an evaluation can be used in the model (e.g., an evaluation of an objective). Our model also supports evaluations of evaluations, which may be able to support attacks on attacks as used in [Chu-Carroll and Carberry2000].

To model full argumentation, however, we need to be able to model a reason or *argument* for or against the proposition. Argumentation allows a reason or argument to be supplied for the attack or support, something which our model does not have at the moment. This addition seems like a natural extension and would widen the range of dialogue that can be handled by the model.

9.1.2 Grounding

Another area which needs improvement is the model of Grounding Acts (GAs). In Chapter 4, we simply took the act types from Conversational Acts [Traum and Hinkelman1992] and defined a single parameter value for them: the Interaction Act (IntAct) which is to be grounded. Although this may be sufficient for several of the acts (e.g., *initiate*, *cancel*), operationalizing the model has shown us that more/different information is needed in some cases.

First, we model separate grounding acts for each of the intended IntActs from the speaker. As an example, we will suppose that agent A utters something to B that has three Grounding Acts as the correct interpretation. Say, however, that the B does not understand the utterance at all (e.g., hears that it was meant for him, but does not understand anything from the content), and says something like “Could you repeat that?” as a response. It is now unclear, using our model, how the communicative intentions of this response should be modeled. It is clear that the Grounding Act is a *ReqRepair* but

how many are there? Right now, our model would say that there should be three *ReqRepairs*, since there were three GAs in A's utterance, but B obviously does not know that, since he was not able to decode the message that far.

A possible solution would be to model grounding acts at each of Clark's [Clark1996] four levels of communication, and at lower levels, only have grounding occur on the utterance as a whole. For example, in the case given above, the communication failure occurs at the signal level, i.e., B knows a signal was sent to him, but does not know what the content was. We could therefore model the response "Can you repeat that?" as a *ReqRepair* at that level (and thus on the signal) or something similar.

Another need to better operationalize the grounding model, is that some GAs need to provide more information than just the IntAct to be grounded.

Again, for example, take the *ReqRepair* act. This time, suppose that A utters "Pick up the block". Suppose that B responds with "Which block?", which would be modeled as a *ReqRepair*, and this time, (we suppose that) B understands exactly what IntActs were meant, just not which block (which would be modeled as a nested resource within the objective). However, simply wrapping these intentions in a *ReqRepair* gives no indication about which *part* of the utterance has been requested to be repaired. Similar examples can be given for *repair* and even *ack* (e.g., for differentiating varying degrees of acknowledgments for different parts of the utterance (cf. [Clark1996]).

9.2 Future Work in Goal Recognition

In this section, we outline next steps for work in statistical goal recognition.

9.2.1 Further Testing with New Corpora

As discussed in Chapter 6, we are aware of only very few corpora for plan recognition. We have tested our flat recognizer on the Linux corpus and the Monroe corpus, and our

hierarchical recognizer only on the Monroe corpus. An important next step would be to evaluate the recognizers in different domains. This is especially true for the hierarchical recognizer, as the recognition results are based on an artificially generated corpus.

Especially interesting to us would be to gather a human corpus in a domain similar to Monroe and annotate it (by hand, most likely) with plan information. The creation and use of this corpus could help answer several open questions from our work, including: How difficult is it to hand annotate a corpus with plan information? What differences are exhibited in human versus artificial corpora? How well does our hierarchical recognizer perform on human-produced data? How well can we recognize human goals with a recognizer trained on artificially generated data?

9.2.2 Conditioning on Parameter Values

Another important next step will be to remove one of the the simplifying assumptions we made in Chapter 7, namely, that the probability of a goal schema is independent of action parameter values, given their action schema. As we mentioned in Chapter 7, this is not always the case, as the goal schema can very much depend on what the action parameter values are.

The main reason for making this assumption was data sparsity. Right now, both the flat and hierarchical schema recognizers use a fairly straightforward bigram model over action schemas. If we do not make this independence assumption, however, we need to introduce action parameter values into the equation, which would lead to an explosion in the number of possible instantiated actions in the domain (in the worst case, exponential in the number of parameter positions and objects in the domain).

There are several potential solutions to this that could be tried. The first would be to use some sort of abstraction backoff for getting conditional probabilities. The idea would be similar to the n-gram backoff we use in the recognizer right now. The recognizer would first look for the most specific conditional probability (i.e., the bigram

of action schemas and their actual parameter values). If this was not found (or was not found enough times) in the training data, then the recognizer would look for the probability of some sort of abstraction of that bigram. For example, parameter values could be abstract to their domain types (e.g., vehicle or person) and then those conditional probabilities could be searched. By using a domain ontology, abstraction backoff could happen until at the end, it abstracts to just using the action schemas themselves. This would theoretically give us more information where it is available in the training data, but give us a backoff to what is happening now.

Another possibility would be to use data mining techniques (cf. [Zaki, Lesh, and Ogihara2000]) to automatically identify cases in the data where action parameter values are particularly helpful in discriminating goal schema values. The intuition here is that, in our experience, in many cases, the action parameter values really do seem to be independent from the goal schema. However, there are some easily identifiable cases where they are nearly always a distinguishing factor. Discovering these would allow the recognizer to take those into account only where helpful.

9.2.3 Problem Solving Recognition

Finally, we believe that an important area of future research (brought out especially from the focus of this thesis) will be the generalization of plan recognition to what we will call *problem solving recognition*. Plan recognition is typically defined as the recognition of an agent's plan, given observed actions. This definition, however, implicitly makes the same assumption many plan-based dialogue models do (as discussed in Chapter 2) namely, that an agent first creates a plan, and then executes it.

Of course this is not always the case, and we would argue that there are many domains in which this is usually *not* the case. We believe if we want to model real agents from observations, we need to recognize the agent's problem-solving activity itself. This would mean recognizing the agent's current problem-solving state, which could

then change from timestep to timestep. There would no longer be a single *plan* data object attached to a plan session, rather, a post hoc view of a plan session would reveal a trace of the agent's problem-solving state over the session. The agent may have had several (partial or full) plans over the session, which may have been expanded or revised (or scrapped) as time passes. This would also model shifts in execution of different plans for different objectives, and even phenomena like goal abandonment. (It could be very useful to know when an agent has abandoned a goal without accomplishing it.)

As this is a very new area, much work is needed here. However, as a possible extention to our work, we have considered the possibility of using an artificial agent to create a *problem-solving labeled corpus* which could then give us information about not only hierarchical goal structure over time but also could be used to train a recognizer to predict when phenomena like replanning or goal abandonment have occurred.

9.3 Future Work in Agent-based Dialogue Systems

In this section, we describe directions of future work needed for supporting agent-based dialogue systems. We discuss these by subsystem: starting first with interpretation, then moving to behavior, and finally discussing generation.

9.3.1 Interpretation

An agent-based dialogue system needs to be able to convert input language into the corresponding communicative intentions, i.e., the instantiated grounding acts from Chapter 4. This, of course, is highly context dependent, thus an intention recognizer would need to take the dialogue state into account, as well as the semantics of the utterance.

We believe a good starting point for an agent-based intention recognizer will be the basic ideas of a plan-based recognizer (e.g., [Lochbaum1998; Chu-Carroll and Carberry2000]). However, as we discussed in Chapter 2, plan-based intention recogniz-

ers are only able recognize intentions based on their own dialogue model. Thus, a plan-based intention recognizer would need to be augmented to recognize the range of collaborative problem-solving activity which we need to for agent-based dialogue.

Most intention recognition algorithms are based on plan recognition, and are therefore not scalable. To provide scalability, the next step would be to incorporate our hierarchical goal recognizer into the intention recognizer. Although the goal recognizer cannot recognize intentions on its own, it does provide a fast way to narrow down the search space (e.g., by an n-best prediction) to allow for a slower, symbolic recognition algorithm to perform the recognition of the actual intention.

9.3.2 Behavior

Once the user's intentions have successfully been recognized, the dialogue model described in Chapter 4 defines in which way they update the dialogue state. Given an updated dialogue state, we need a behavioral component which can make decisions about what to do next, both in terms of interaction with the world (through sensing and acting) as well as interaction through communication.

To provide a truly agent-based system, we want to use an autonomous agent to control behavior. Work in the agents field has made progress in designing agents which act based on their beliefs about the world, their desires for how the world should be, and intentions for action ([Rao and Georgeff1991]), but these agents typically do not know how to collaborate with others. Research needs to be done in programming such agents to take problem-solving obligations into account in decision-making. Such agents also need to be able to generate their own communicative intentions which can be passed on to generation.

9.3.3 Generation

The final area of needed research for agent-based systems is in generation, particularly in the area of content planning, where communicative intentions generated by the behavioral subsystem are converted into language to be communicated with the user. This is perhaps the most wide-open field, as most research in language generation for dialogue has taken high-level semantic forms as input, instead of communicative intentions.

Bibliography

[Agre and Horswill1992] Agre, P. and I. Horswill. 1992. Cultural support for improvisation. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, pages 363–368.

[Albrecht, Zukerman, and Nicholson1998] Albrecht, David W., Ingrid Zukerman, and Ann E. Nicholson. 1998. Bayesian models for keyhole plan recognition in an adventure game. *User Modeling and User-Adapted Interaction*, 8:5–47.

[Alexandersson et al.1998] Alexandersson, Jan, Bianka Buschbeck-Wolf, Tsutomu Fujinami, Michael Kipp, Stephan Kock, Elisabeth Maier, Norbert Reithinger, Birte Schmitz, and Melanie Siegel. 1998. Dialogue acts in VERBMOBIL-2 second edition. Verbmobil Report 226, DFKI Saarbrücken, Universität Stuttgart, TU Berlin, Universität des Saarlandes, July.

[Allen et al.2000] Allen, J., D. Byron, M. Dzikovska, G. Ferguson, L. Galescu, and A. Stent. 2000. An architecture for a generic dialogue shell. *Journal of Natural Language Engineering special issue on Best Practices in Spoken Language Dialogue Systems Engineering*, 6(3):1–16, December.

[Allen1983] Allen, James. 1983. Recognizing intentions from natural language utterances. In M. Brady and R. C. Berwick, editors, *Computational Models of Discourse*. MIT Press, pages 107–166.

[Allen, Blaylock, and Ferguson2002] Allen, James, Nate Blaylock, and George Ferguson. 2002. A problem-solving model for collaborative agents. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 774–781, Bologna, Italy, July 15-19. ACM Press.

[Allen and Core1997] Allen, James and Mark Core. 1997. Draft of DAMSL: Dialog act markup in several layers. Available at http://www.cs.rochester.edu/research/cisd/resources/damsl/, October.

[Allen, Ferguson, and Stent2001] Allen, James, George Ferguson, and Amanda Stent. 2001. An architecture for more realistic conversational systems. In *Proceedings of Intelligent User Interfaces 2001 (IUI-01)*, pages 1–8, Santa Fe, NM, January.

[Allen1979] Allen, James F. 1979. A plan-based approach to speech act recognition. Technical Report 131/79, University of Toronto. PhD thesis.

[Allen et al.2001] Allen, James F., Donna K. Byron, Myroslava Dzikovska, George Ferguson, Lucian Galescu, and Amanda Stent. 2001. Towards conversational human-computer interaction. *AI Magazine*, 22(4):27–37.

[Allen and Perrault1980] Allen, James F. and C. Raymond Perrault. 1980. Analyzing intention in utterances. *Artificial Intelligence*, 15(3):143–178.

[Appelt and Pollack1991] Appelt, Douglas E. and Martha E. Pollack. 1991. Weighted abduction for plan ascription. *User Modeling and User-Adapted Interaction*, 2:1–25.

[Ardissono, Boella, and Lesmo1996] Ardissono, Liliana, Guido Boella, and Leonardo Lesmo. 1996. Recognition of problem-solving plans in dialogue interpretation. In *Proceedings of the Fifth International Conference on User Modeling*, pages 195–197, Kailua-Kona, Hawaii, January.

[Ashbrook and Starner2003] Ashbrook, Daniel and Thad Starner. 2003. Using GPS to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing*, 7(5).

[Austin1962] Austin, J. L. 1962. *How to Do Things with Words*. Harvard University Press, Cambridge, Massachusetts.

[Azarewicz et al.1986] Azarewicz, Jerome, Glenn Fala, Ralph Fink, and Christof Heithecker. 1986. Plan recognition for airborne tactical decision making. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 805–811, Philadelphia.

[Bauer1994] Bauer, Mathias. 1994. Quantitative modeling of user preferences for plan recognition. In B. Goodman, A. Kobsa, and D. Litman, editors, *Proceedings of the Fourth International Conference on User Modeling (UM94)*, pages 73–78, Hyannis, Massachusetts, August. MITRE Corporation.

[Bauer1995] Bauer, Mathias. 1995. A Dempster-Shafer approach to modeling agent preferences for plan recognition. *User Modeling and User-Adapted Interaction*, 5(3–4):317–348.

[Bauer1996a] Bauer, Mathias. 1996a. Acquisition of user preferences for plan recognition. In *Proceedings of the Fifth International Conference on User Modeling*, pages 105–112, Kailua-Kona, Hawaii, January.

[Bauer1996b] Bauer, Mathias. 1996b. Machine learning for user modeling and plan recognition. In *Working Notes of the International Conference on Machine Learning Workshop ML Meets HCI*, Bari, Italy, July 3.

[Bauer1998] Bauer, Mathias. 1998. Acquisition of abstract plan descriptions for plan recognition. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 936–941, Madison, WI, July.

[Bauer and Paul1993] Bauer, Mathias and Gabriele Paul. 1993. Logic-based plan recognition for intelligent help systems. In Christer Bäckström and Erik Sandewall, editors, *Current Trends in AI Planning: EWSP '93 — Second European Workshop on Planning*, Frontiers in Artificial Intelligence and Applications. IOS Press, Vadstena, Sweden, December, pages 60–73. Also DFKI Research Report RR-93-43.

[Blaylock2002] Blaylock, Nate. 2002. Managing communicative intentions in dialogue using a collaborative problem-solving model. Technical Report 774, University of Rochester, Department of Computer Science, April.

[Blaylock and Allen2003] Blaylock, Nate and James Allen. 2003. Corpus-based, statistical goal recognition. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1303–1308, Acapulco, Mexico, August 9–15.

[Blaylock and Allen2004] Blaylock, Nate and James Allen. 2004. Statistical goal parameter recognition. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04)*, pages 297–304, Whistler, British Columbia, June 3–7. AAAI Press.

[Blaylock and Allen2005a] Blaylock, Nate and James Allen. 2005a. A collaborative problem-solving model of dialogue. In *Proceedings of the SIGdial Workshop on Discourse and Dialog*, Lisbon, September 2–3. To appear.

[Blaylock and Allen2005b] Blaylock, Nate and James Allen. 2005b. Generating artificial corpora for plan recognition. In Liliana Ardissono, Paul Brna, and Antonija Mitrovic, editors, *User Modeling 2005*, number 3538 in Lecture Notes in Artificial Intelligence. Springer, Edinburgh, July 24–29, pages 179–188.

[Blaylock and Allen2005c] Blaylock, Nate and James Allen. 2005c. Recognizing instantiated goals using statistical methods. In Gal Kaminka, editor, *Workshop on Modeling Others from Observations (MOO-2005)*, pages 79–86, Edinburgh, July 30.

[Blaylock, Allen, and Ferguson2002] Blaylock, Nate, James Allen, and George Ferguson. 2002. Synchronization in an asynchronous agent-based architecture for

dialogue systems. In *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialog*, Philadelphia, July.

[Blaylock, Allen, and Ferguson2003] Blaylock, Nate, James Allen, and George Ferguson. 2003. Managing communicative intentions with collaborative problem solving. In Jan van Kuppevelt and Ronnie W. Smith, editors, *Current and New Directions in Discourse and Dialogue*, volume 22 of *Kluwer Series on Text, Speech and Language Technology*. Kluwer, Dordrecht, pages 63–84.

[Blum and Furst1997] Blum, Avrim L. and Merrick L. Furst. 1997. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

[Bohlin et al.1999] Bohlin, Peter, Johan Bos, Staffan Larsson, Ian Lewin, Colin Matheson, and David Milward. 1999. Survey of existing interactive systems. Deliverable D1.3, EU Project TRINDI, February.

[Bohus and Rudnicky2003] Bohus, Dan and Alexander I. Rudnicky. 2003. RavenClaw: Dialog management using hierarchical task decomposition and an expectation agenda. In *Proceedings of Eurospeech-2003*, Geneva, Switzerland.

[Bui2002] Bui, Hung H. 2002. Efficient approximate inference for online probabilistic plan recognition. Technical Report 1/2002, School of Computing, Curtin University of Technology.

[Bui2003] Bui, Hung H. 2003. A general model for online probabilistic plan recognition. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, Acapulco, Mexico, August 9–15.

[Bui, Venkatesh, and West2002] Bui, Hung H., Svetha Venkatesh, and Geoff West. 2002. Policy recognition in the Abstract Hidden Markov Model. *Journal of Artificial Intelligence Research*, 17:451–499.

[Carberry1983] Carberry, Sandra. 1983. Tracking user goals in an information-seeking environment. In *Proceedings of the Third National Conference on Artificial Intelligence*, pages 59–63, Washington, D.C.

[Carberry1987] Carberry, Sandra. 1987. Pragmatic modeling: Toward a robust natural language interface. *Computational Intelligence*, 3:117–136.

[Carberry1990a] Carberry, Sandra. 1990a. Incorporating default inferences into plan recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 471–478, Boston, July 29 – August 3. AAAI Press.

[Carberry1990b] Carberry, Sandra. 1990b. *Plan Recognition in Natural Language Dialogue*. ACL-MIT Press Series on Natural Language Processing. MIT Press.

[Carberry, Kazi, and Lambert1992] Carberry, Sandra, Zunaid Kazi, and Lynn Lambert. 1992. Modeling discourse, problem-solving and domain goals incrementally in task-oriented dialogue. In *Proc. 3rd Int. Workshop on User Modeling*, pages 192–201. Wadern.

[Carletta et al.1997] Carletta, Jean, Amy Isard, Stephen Isard, Jacqueline C. Kowtko, Gwyneth Doherty-Sneddon, and Anne H. Anderson. 1997. The reliability of a dialogue structure coding scheme. *Computational Linguistics*, 23(1):13–31.

[Charniak1997] Charniak, Eugene. 1997. Statistical techniques for natural language parsing. *AI Magazine*, 18(4):33–43.

[Charniak and Goldman1993] Charniak, Eugene and Robert P. Goldman. 1993. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79.

[Chu-Caroll and Brown1997] Chu-Caroll, Jennifer and Michael K. Brown. 1997. Initiative in collaborative interactions — its cues and effects. In S. Haller and S. McRoy, editors, *Working Notes of AAAI Spring 1997 Symposium on Computational Models of Mixed Initiative Interaction*, pages 16–22, Stanford, CA.

[Chu-Carroll2000] Chu-Carroll, Jennifer. 2000. MIMIC: An adaptive mixed initiative spoken dialogue system for information queries. In *Proceedings of the 6th Conference on Applied Natural Language Processing*, pages 97–104.

[Chu-Carroll and Carberry1994] Chu-Carroll, Jennifer and Sandra Carberry. 1994. A plan-based model for response generation in collaborative task-oriented dialogues. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 799–805, Seattle, WA.

[Chu-Carroll and Carberry1995] Chu-Carroll, Jennifer and Sandra Carberry. 1995. Communication for conflict resolution in multi-agent collaborative planning. In V. Lesser, editor, *Proceedings of the First International Conference on Multiagent Systems*, pages 49–56. AAAI Press.

[Chu-Carroll and Carberry1996] Chu-Carroll, Jennifer and Sandra Carberry. 1996. Conflict detection and resolution in collaborative planning. In M. Woodbridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II: Agent Theories, Architectures, and Languages*, number 1037 in Lecture Notes in Artificial Intelligence. Springer-Verlag, pages 111–126.

[Chu-Carroll and Carberry2000] Chu-Carroll, Jennifer and Sandra Carberry. 2000. Conflict resolution in collaborative planning dialogues. *International Journal of Human-Computer Studies*, 53(6):969–1015.

[Clark1996] Clark, Herbert H. 1996. *Using Language*. Cambridge University Press.

[Cohen1978] Cohen, Philip R. 1978. On knowing what to say: Planning speech acts. Technical Report 118, Department of Computer Science, University of Toronto, Ontario, January. PhD thesis.

[Cohen1994] Cohen, Philip R. 1994. Models of dialogue. In T. Ishiguro, editor, *Cognitive Processing for Voice and Vision*. Society of Industrial and Applied Mathematics, pages 181–203.

[Cohen and Levesque1990a] Cohen, Philip R. and Hector J. Levesque. 1990a. Intention is choice with commitment. *Artificial Intelligence*, 42:213–261.

[Cohen and Levesque1990b] Cohen, Philip R. and Hector J. Levesque. 1990b. Persistence, intention, and commitment. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, pages 33–69.

[Cohen and Levesque1990c] Cohen, Philip R. and Hector J. Levesque. 1990c. Rational interaction as the basis for communication. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, pages 221–254.

[Cohen et al.1991] Cohen, Philip R., Hector J. Levesque, José H. T. Nunes, and Sharon L. Oviatt. 1991. Task-oriented dialogue as a consequence of joint activity. In Hozumi Tanaka, editor, *Artificial Intelligence in the Pacific Rim*. IOS Press, Amsterdam, pages 203–208.

[Cohen and Perrault1979] Cohen, Philip R. and C. Raymond Perrault. 1979. Elements of a plan-based theory of speech acts. *Cognitive Science*, 3:177–212. Reprinted in B.J. Grosz, K. Sparck-Jones, and B.L. Webber, editors, Readings in Natural Language Processing, Morgan Kaufmann, Los Altos, 1986 and Reprinted in L. Gasser and M. Huhns, editors, Readings in Distributed Artificial Intelligence, Morgan Kaufmann, Los Altos, 1988.

[Cohen, Perrault, and Allen1982] Cohen, Philip R., C. Raymond Perrault, and James F. Allen. 1982. Beyond question answering. In Wendy G. Lehnert and Martin H. Ringle, editors, *Strategies for Natural Language Processing*. Lawrence Erlbaum Associates, pages 245–274.

[DARPA Knowledge Sharing Initiative, External Interfaces Working Group1993] DARPA Knowledge Sharing Initiative, External Interfaces Working Group. 1993. Specification of the KQML agent-communication language. Working paper, June.

[Davison and Hirsh1997] Davison, Brian D. and Haym Hirsh. 1997. Experiments in UNIX command prediction. Technical Report ML-TR-41, Department of Computer Science, Rutgers University.

[Davison and Hirsh1998] Davison, Brian D. and Haym Hirsh. 1998. Predicting sequences of user actions. In *Notes of the AAAI/ICML 1998 Workshop on Predicting the Future: AI Approaches to Time-Series Analysis*, Madison, Wisconsin.

[Di Eugenio et al.1997] Di Eugenio, Barbara, Pamela W. Jordan, Richmond H. Thomason, and Johanna D. Moore. 1997. Reconstructed intentions in collaborative problem solving dialogues. In *Working Notes of AAAI Fall Symposium on Communicative Action in Humans and Machines*, Cambridge, Massachusetts, November.

[Ferguson and Allen1998] Ferguson, George and James F. Allen. 1998. TRIPS: An intelligent integrated intelligent problem-solving assistant. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 567–573, Madison, WI, July.

[Fine, Singer, and Tishby1998] Fine, Shai, Yoram Singer, and Naftali Tishby. 1998. The Hierarchical Hidden Markov Model: Analysis and applications. *Machine Learning*, 32:41–62.

[Geib and Goldman2001] Geib, Christopher W. and Robert P. Goldman. 2001. Plan recognition in intrusion detection systems. In *2nd DARPA Information Survivability Conference and Exposition (DISCEX-II 2001)*, pages 46–55, Anaheim, California, June 12–14.

[Goldman, Geib, and Miller1999] Goldman, Robert P., Christopher W. Geib, and Christopher A. Miller. 1999. A new model of plan recognition. In *Uncertainty in Artificial Intelligence: Proceedings of the Fifteenth Conference (UAI-1999)*, pages 245–254, San Francisco, CA. Morgan Kaufmann Publishers.

[Grice1957] Grice, H. P. 1957. Meaning. *Philosophical Review*, 66(3):377–388.

[Grice1969] Grice, H. Paul. 1969. Utterer's meaning and intention. *Philosophical Review*, 78(2):147–177.

[Grice1975] Grice, H. Paul. 1975. Logic and conversation. In P. Cole and J. L. Morgan, editors, *Speech Acts*, volume 3 of *Syntax and Semantics*. Academic Press, New York, pages 41–58.

[Gross, Allen, and Traum1992] Gross, Derek, James Allen, and David Traum. 1992. The Trains 91 dialogues. TRAINS Technical Note 92-1, University of Rochester, Department of Computer Science.

[Grosz and Sidner1986] Grosz, Barbara and Candace Sidner. 1986. Attention, intention, and the structure of discourse. *Computational Linguistics*, 12(3):175–204.

[Grosz1981] Grosz, Barbara J. 1981. Focusing and description in natural language dialogues. In A. Joshi, B. Webber, and I. Sag, editors, *Elements of Discourse Understanding*. Cambridge University Press, New York, New York, pages 84–105.

[Grosz and Kraus1996] Grosz, Barbara J. and Sarit Kraus. 1996. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357.

[Grosz and Kraus1999] Grosz, Barbara J. and Sarit Kraus. 1999. The evolution of SharedPlans. In A. Rao and M. Wooldridge, editors, *Foundations and Theories of Rational Agency*. Kluwer, pages 227–262.

[Grosz and Sidner1990] Grosz, Barbara J. and Candace L. Sidner. 1990. Plans for discourse. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, pages 417–444.

[Hansen, Novick, and Sutton1996] Hansen, Brian, David G. Novick, and Stephen Sutton. 1996. Systematic design of spoken prompts. In *Conference on Human Factors in Computing Systems (CHI'96)*, pages 157–164, Vancouver, British Columbia, April.

[Hong2001] Hong, Jun. 2001. Goal recognition through goal graph analysis. *Journal of Artificial Intelligence Research*, 15:1–30.

[Horvitz and Paek1999] Horvitz, Eric and Tim Paek. 1999. A computational architecture for conversation. In *Proceedings of the Seventh International Conference on User Modeling*, pages 201–210, Banff, Canada, June. Springer-Verlag.

[Huber, Durfee, and Wellman1994] Huber, Marcus J., Edmund H. Durfee, and Michael P. Wellman. 1994. The automated mapping of plans for plan recognition. In R. L. de Mantaras and D. Poole, editors, *UAI94 - Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 344–351, Seattle, Washington. Morgan Kaufmann.

[Jurafsky and Martin2000] Jurafsky, Daniel and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall.

[Kautz1990] Kautz, Henry. 1990. A circumscriptive theory of plan recognition. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, pages 105–134.

[Kautz1991] Kautz, Henry. 1991. A formal theory of plan recognition and its implementation. In J. Allen, H. Kautz, R. Pelavin, and J. Tenenberg, editors, *Reasoning about Plans*. Morgan Kaufman, San Mateo, CA, pages 69–125.

[Kautz and Allen1986] Kautz, Henry and James Allen. 1986. Generalized plan recognition. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 32–37, Philadelphia.

[Kautz1987] Kautz, Henry A. 1987. A formal theory of plan recognition. Technical Report 215, University of Rochester, Department of Computer Science. PhD thesis.

[Kellner1998] Kellner, Andreas. 1998. Initial language models for spoken dialogue systems. In *Proceedings of ICASSP'98*, pages 185–188, Seattle, Washington.

[Lambert1993] Lambert, Lynn. 1993. Recognizing complex discourse acts: A tripartite plan-based model of dialogue. Technical Report 93-19, University of Delaware, Department of Computer and Information Sciences, Newark, Delaware, May. PhD thesis.

[Lambert and Carberry1991] Lambert, Lynn and Sandra Carberry. 1991. A tripartite plan-based model of dialogue. In *Proceedings of the 29th ACL*, pages 47–54, Berkeley, CA, June.

[Lamel et al.2000] Lamel, L., S. Rosset, J. L. Gauvain, S. Bennacef, M. Garnier-Rizet, and B. Protus. 2000. The LIMSI ARISE system. *Speech Communication*, 31(4):339–354, August.

[Larsson2002] Larsson, Staffan. 2002. Issues under negotiation. In *Proceedings of the 3rd SIGdial Workshop on Discourse and Dialog*, pages 103–112, Philadelphia, July.

[Lemon, Gruenstein, and Peters2002] Lemon, Oliver, Alexander Gruenstein, and Stanley Peters. 2002. Collaborative activities and multi-tasking in dialogue systems: Towards natural language with robots. *Traitement Automatique des Langues (TAL)*, 43(2):131–154.

[Lesh1998] Lesh, Neal. 1998. *Scalable and Adaptive Goal Recognition*. Ph.D. thesis, University of Washington.

[Lesh and Etzioni1995a] Lesh, Neal and Oren Etzioni. 1995a. Insights from machine learning for plan recognition. In M. Bauer, editor, *IJCAI 95 Workshop on The Next Generation of Plan Recognition Systems: Challenges for and Insight from Related Areas of AI (Working Notes)*, pages 78–83, Montreal, Canada.

[Lesh and Etzioni1995b] Lesh, Neal and Oren Etzioni. 1995b. A sound and fast goal recognizer. In *IJCAI95 - Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1704–1710, Montreal, Canada.

[Lesh and Etzioni1996] Lesh, Neal and Oren Etzioni. 1996. Scaling up goal recognition. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR96)*, pages 178–189.

[Levesque, Cohen, and Nunes1990] Levesque, H., P. Cohen, and J. Nunes. 1990. On acting together. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 94–99, Boston, July 29 – August 3. AAAI Press.

[Litman1985] Litman, Diane J. 1985. Plan recognition and discourse analysis: An integrated approach for understanding dialogues. Technical Report TR170, University of Rochester, Department of Computer Science. PhD thesis.

[Litman1986] Litman, Diane J. 1986. Understanding plan ellipsis. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 619–624, Philadelphia.

[Litman and Allen1987] Litman, Diane J. and James F. Allen. 1987. A plan recognition model for subdialogues in conversations. *Cognitive Science*, 11(2):163–200.

[Litman and Allen1990] Litman, Diane J. and James F. Allen. 1990. Discourse processing and commonsense plans. In P. R. Cohen, J. Morgan, and M. Pollack, editors, *Intentions in Communication*. MIT Press, Cambridge, MA, pages 365–388.

[Lochbaum1998] Lochbaum, Karen E. 1998. A collaborative planning model of intentional structure. *Computational Linguistics*, 24(4):525–572.

[Lochbaum, Grosz, and Sidner2000] Lochbaum, Karen E., Barbara J. Grosz, and Candace L. Sidner. 2000. Discourse structure and intention recognition. In Robert Dale, Hermann Moisl, and Harold Sommers, editors, *Handbook of Natural Language Processing*. Marcel Dekker, New York, pages 123–146.

[Mann and Thompson1987] Mann, William C. and Sandra A. Thompson. 1987. Rhetorical structure theory: A theory of text organization. In L. Polanyi, editor, *The Structure of Discourse*. Ablex Publishing Corporation.

[McRoy1998] McRoy, Susan W. 1998. Achieving robust human-computer communication. *International Journal of Human-Computer Studies*, 48:681–704.

[Murphy and Paskin2001] Murphy, Kevin P. and Mark A. Paskin. 2001. Linear time inference in hierarchical HMMs. In *NIPS-01*.

[Nau et al.2003] Nau, Dana, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. 2003. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404.

[Patterson et al.2003] Patterson, Donand J., Lin Liao, Dieter Fox, and Henry Kautz. 2003. Inferring high-level behavior from low-level sensors. In *Fifth Annual Conference on Ubiquitous Computing (UBICOMP 2003)*, Seattle, Washington.

[Pollack1986] Pollack, Martha. 1986. Inferring domain plans in question-answering. Technical Report MS-CIS-86-40 LINC LAB 14, University of Pennsylvania, May. PhD thesis.

[Pollard and Sag1994] Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Studies in Contemporary Linguistics. University of Chicago Press, Chicago.

[Pynadath1999] Pynadath, David V. 1999. *Probabilistic Grammars for Plan Recognition*. Ph.D. thesis, University of Michigan, Department of Computer Science and Engineering.

[Pynadath and Wellman1995] Pynadath, David. V. and Michael. P. Wellman. 1995. Accounting for context in plan recognition, with application to traffic monitoring. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 472–481, Montreal, Canada. Morgan Kaufmann.

[Pynadath and Wellman2000] Pynadath, David V. and Michael P. Wellman. 2000. Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI-2000)*, pages 507–514, Stanford, CA, June.

[Ramshaw1989] Ramshaw, Lance A. 1989. A metaplan model for problem-solving discourse. In *Proceedings of the Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 35–42, Manchester, England.

[Ramshaw1991] Ramshaw, Lance A. 1991. A three-level model for plan exploration. In *Proceedings of the 29th ACL*, pages 39–46, Berkeley, CA, June.

[Ramshaw1989] Ramshaw, Lance Arthur. 1989. Pragmatic knowledge for resolving ill-formedness. Technical Report 89-18, University of Delaware, Newark, Delaware, June. PhD thesis.

[Rao and Georgeff1995] Rao, A. and M. Georgeff. 1995. BDI agents: From theory to practice. In V. Lesser, editor, *Proceedings of the First International Conference on Multiagent Systems*. AAAI Press.

[Rao and Georgeff1991] Rao, Anand S. and Michael P. Georgeff. 1991. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Principles of Knowledge Representation and Reasoning*, pages 473–484, Cambridge, Massachusetts, April 22-25. Morgan Kaufmann. Also available as AAII Technical Note 14.

[Rayner, Hockey, and James2000] Rayner, Manny, Beth Ann Hockey, and Frankie James. 2000. A compact architecture for dialogue management based on scripts and meta-outputs. In *Proceedings of the 6th Conference on Applied Natural Language Processing*.

[Rich and Sidner1998] Rich, Charles and Candace L. Sidner. 1998. COLLAGEN: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction*, 8(3–4):315–350. Also available as MERL Technical Report 97-21a.

[Rich, Sidner, and Lesh2001] Rich, Charles, Candace L. Sidner, and Neal Lesh. 2001. COLLAGEN: Applying collaborative discourse theory to human-computer interaction. *AI Magazine*, 22(4):15–25. Also available as MERL Tech Report TR-2000-38.

[Rudnicky et al.1999] Rudnicky, A. I., E. Thayer, P. Constantinides, C. Tchou, R. Shern, K. Lenzo, W. Xu, and A. Oh. 1999. Creating natural dialogs in the Carnegie Mellon Communicator system. In *Proceedings of the 6th European Conference on Speech Communication and Technology (Eurospeech-99)*, pages 1531–1534, Budapest, Hungary, September.

[Sadek and De Mori1998] Sadek, David and Renato De Mori. 1998. Dialogue systems. In Renato De Mori, editor, *Spoken Dialogues with Computers*, Signal Processing and Its Applications. Academic Press, London, pages 523–561.

[Schmidt, Sridharan, and Goodson1978] Schmidt, C. F., N. S. Sridharan, and J. L. Goodson. 1978. The plan recognition problem: An intersection of psychology and artificial intelligence. *Artificial Intelligence*, 11:45–83.

[Searle1975] Searle, John R. 1975. Indirect speech acts. In P. Cole and J. L. Morgan, editors, *Speech Acts*, volume 3 of *Syntax and Semantics*. Academic Press, New York, pages 59–82.

[Seneff and Polifroni2000] Seneff, Stephanie and Joseph Polifroni. 2000. Dialogue management in the Mercury flight reservation system. In *Proceedings of ANLP/NAACL-2000 Workshop on Conversational Systems*, Seattle, Washington, May.

[Sidner1994] Sidner, Candace L. 1994. An artificial discourse language for collaborative negotiation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 814–819, Seattle, WA. Also available as Lotus Technical Report 94-09.

[Sidner and Israel1981] Sidner, Candace L. and David J. Israel. 1981. Recognizing intended meaning and speakers' plans. In *Proceedings of International Joint Conference on Artificial Intelligence*, pages 203–208, Vancouver, B.C.

[Sidner1994] Sidner, Candy. 1994. Negotiation in collaborative activity: A discourse analysis. *Knowledge-Based Systems*, 7(4):265–267. Also available as Lotus Technical Report 94-10.

[Stent2000] Stent, Amanda J. 2000. The Monroe corpus. Technical Report 728, University of Rochester, Department of Computer Science, March. Also Technical Note 99-2.

[The Foundation for Intelligent Physical Agents2002] The Foundation for Intelligent Physical Agents. 2002. FIPA Request Interaction Protocol specification. http://www.fipa.org/specs/fipa00026/SC00026H.html, December.

[Traum1994] Traum, David R. 1994. A computational theory of grounding in natural language conversation. Technical Report 545, University of Rochester, Department of Computer Science, December. PhD Thesis.

[Traum2000] Traum, David R. 2000. 20 questions for dialogue act taxonomies. *Journal of Semantics*, 17(1):7–30.

[Traum and Hinkelman1992] Traum, David R. and Elizabeth A. Hinkelman. 1992. Conversation acts in task-oriented spoken dialogue. *Computational Intelligence*, 8(3):575–599. Also available as University of Rochester Department of Computer Science Technical Report 425.

[Vilain1990] Vilain, Marc. 1990. Getting serious about parsing plans: a grammatical analysis of plan recognition. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 190–197, Boston, July 29 – August 3. AAAI Press.

[Wilensky1983] Wilensky, Robert. 1983. *Planning and Understanding: A Computational Approach to Human Reasoning*. Addison-Wesley, Reading, Massachusetts.

[Wooldridge and Jennings1999] Wooldridge, Michael and Nicholas R. Jennings. 1999. The cooperative problem-solving process. *Journal of Logic and Computation*, 9(4):563–592.

[Wu2003] Wu, Huadong. 2003. *Sensor Data Fusion for Context-Aware Computing Using Dempster-Shafer Theory*. Ph.D. thesis, Carnegie Mellon University, Robotics Institute, December.

[Zaki, Lesh, and Ogihara2000] Zaki, Mohammed J., Neal Lesh, and Mistunori Ogihara. 2000. PLANMINE: Predicting plan failures using sequence mining. *Artificial Intelligence Review*, 14(6):421–446, December. special issue on the Application of Data Mining.

[Zue et al.2000] Zue, Victor, Stephanie Seneff, James Glass, Joseph Polifrani, Christine Pao, Timothy J. Hazen, and Lee Hetherington. 2000. JUPITER: A telephone-based conversational interface for weather information. *IEEE Transactions on Speech and Audio Processing*, 8(1):100–112, January.

A Instructions Given to Users in the Linux Corpus Collection

We are studying how people perform tasks in Linux. We will give you a series of tasks to complete. In each case, we will record the commands you use (and their results). By continuing, you agree to let us do this recording and use it for further study and/or publications. It will in no way be used to personally identify you.

Each task should take no more than a few minutes at most. You are free to do as many tasks as you like and you may quit at any time.

INSTRUCTIONS

You will be given a task to complete in Linux. When you have successfully completed the task, use the command ’success’ to indicate so. If, at any time, you wish to give up, use ’fail’. Note: the system is not actually checking to see if you accomplished the task or not. It just believes you when you say ’success’ or ’fail’. Use the command ’help’ if you ever need any.

You may perform the task any way you like. However, please follow the following rules:

1. Do everything in the current shell. Don’t invoke new shells (tcsh, rsh, etc.) or do stuff in another program (like emacs). It prevents the program from monitoring

your activity.

2. Don't use scripts (awk, perl, sh, ...)

3. Use one command per line, don't use pipes '|' or commands with other commands embedded in them, (e.g., with ';' or backticks '''). Also, it's ok to use 'find' but not 'find -exec'

4. For each session, you will be assigned a randomly generated directory called: /u/blaylock/Experiment/Playground/username_time (where 'username_time' will be your username and the current time). Please stay within that subdirectory tree (i.e., pretend like /u/blaylock/Experiment/Playground/username_time is /)

5. Use only standard programs. Don't use any shell scripts/programs installed in personal user accounts.

6. Arrows, command completion, and command editing don't work. Sorry.

Remember, there is nowhere you need to put an 'answer' for the task. Simply type 'success' if you accomplished the task, or 'fail' if you are giving up.

The current directory is `dirname`, please treat this as your root directory.

B Goal Schemas in the Linux Corpus

There were 19 goal schemas used in the Linux corpus. Table B.1 shows each schema, along with the template used to generate its English description and its assigned a priori probability.

Note that the English description includes parameters in the form $1, $2, etc. which correspond to the first, second, etc. parameter in the goal schema. In the corpus collection, these variables were instantiated with the value of the actual parameter and then the text was shown to the subject.

`Goal Schema` **English Description**	**Prob.**
`find-file-by-attr-name-exact(filename)` find a file named '$1'	0.091
`find-file-by-attr-name-ext(extension)` find a file that ends in '.$1'	0.055
`find-file-by-attr-name-stem(stem)` find a file that begins with '$1'	0.055
`find-file-by-attr-date-modification-exact(date)` find a file that was last modified $1	0.055
`compress-dirs-by-attr-name(dirname)` compress all directories named '$1'	0.055
`compress-dirs-by-loc-dir(dirname)` compress all subdirectories in directories named '$1'	0.055
`know-filespace-usage-file(filename)` find out how much filespace file '$1' uses	0.073
`know-filespace-usage-partition(partition-name)` find out how much filespace is used on filesystem '$1'	0.055
`know-filespace-free(partition-name)` find out how much filespace is free on filesystem '$1'	0.036
`determine-machine-connected-alive(machine-name)` find out if machine '$1' is alive on the network	0.036
`create-file(filename,dirname)` create a file named '$1' in a (preexisting) directory named '$2'	0.073
`create-dir(create-dirname,loc-dirname)` create a subdirectory named '$1' in a (preexisting) directory named '$2'	0.036
`remove-files-by-attr-name-ext(extention)` delete all files ending in '.$1'	0.036
`remove-files-by-attr-size-gt(numbytes)` delete all files which contain more than $1 bytes	0.018
`copy-files-by-attr-name-ext(extention,dirname)` copy all files ending in '.$1' to a (preexisting) directory named '$2'	0.018
`copy-files-by-attr-size-lt(numbytes,dirname)` copy all files containing less than $1 bytes to a (preexisting) directory named '$2'	0.018
`move-files-by-attr-name-ext(extention,dirname)` move all files ending in '.$1' to a (preexisting) directory named '$2'	0.091
`move-files-by-attr-name-stem(stem,dirname)` move all files beginning with '$1' to a (preexisting) directory named '$2'	0.073
`move-files-by-attr-size-lt(numbytes,dirname)` move all files containing less than $1 bytes to a (preexisting) directory named '$2'	0.073

Table B.1: Goal Schemas in the Linux Corpus

C Action Schemas in the Linux Corpus

We discuss here some of the issues in converting raw Linux command strings into parameterized actions. We first discuss some of the general issues encountered and then discuss the action schemas themselves.

C.1 General Issues for Conversion

The following describes some of the general difficulties we encountered in mapping the Linux domain onto actions. It is important to note that our goal in this project was not to write a general-purpose action-description language for Linux, rather to test a theory of goal recognition, thus some of our choices were pragmatic rather than principled.

Flags Linux uses command flags (e.g., `-l`) in two different ways: to specify unordered parameters and to change the command functionality. The former is fairly easy to handle. The latter, however, is more difficult. It would be possible to treat each command/flags combination as a separate command. However, many commands have various flags, which may be used in various combinations, which would likely lead to a data sparseness problem.

We currently just ignore all command functionality flags. The action schema name used is just the 'command name' of the Linux command (e.g., `ls` from `ls -l -a`). One option would be to form a sort of multiple-inheritance abstraction hierarchy of commands and their flags (e.g., `ls -l -a` inherits from `ls -l` and `ls -a`), although we leave this to future work.

Optional Parameters Treating various modes of commands as one command expands the number of possible parameters for each command. For example, `find` can take the `-size` parameter to search for a file of a certain size, or `-name` to search for a certain name. These parameters can be used together, separately, or not at all.

To deal with this problem, each action schema has a parameter for each *possible* parameter value, but parameter values are allowed to be blank.

Lists of Parameters Many commands actually allow for a list of parameters (usually for their last parameter). The command `ls`, for example, allows a list of filenames or directory names. Rather than handle lists in special ways, we treat this as multiple instances of the action happening at the same timestep (one instance for each parameter in the list).

Filenames and Paths As can be seen below in the action schemas, many commands have both a `path` and a `prepath` parameter. Because our parameter recognizer uses the action parameter values to help predict goal's parameter values, it is necessary that the corresponding value be found in the action parameter where possible. Paths were especially difficult to handle in Linux because they can conceptually be thought of as a *list* of values — namely each subdirectory name in the path as well as a final subdirectory name (in the case that the path refers to a directory) or a filename (in the case it refers to a file). In a complex path, the parameter value was often the last item in the path.

As a solution, we separated each path into a `path` and a `prepath`. The `path` was the last item on the original path, whereas the `prepath` contained the string of the rest of the original path (even if it had more than one subdirectory in it).

As an example, consider the command `cd dir1/dir2/file.txt` which contains a complex path. In this case, it would translate into the following action in our corpus: `cd(dir1/dir2,file.txt)`. This way, the argument `file.txt` becomes a separate parameter value, and thus accessible to the parameter recognizer.

Wildcards How to handle wildcards (* and ?) was another issue. In Linux, filenames containing wildcards are expanded to a list of all matching file and directory names in the working directory. However, that list was not readily available from the corpus. Furthermore, even if there is not match to expand to, we would like to be able to tell that a command `ls *.ps` is looking for an extension `ps`. Our solution was to simply delete all wildcards from filenames.

Current and Parent Directories The special filenames `.` and `..` refer to the current working directory and its parent, respectively. The referents of these were not readily available from the corpus, and leaving them as `.` and `..` made them look like the same parameter value, even though the actual referent was changing (for example when a `cd` was executed).

For each goal session, we rename `.` and `..` to `*dot[num]*` and `*dotdot[num]*`, where `[num]` is the number of `cd` commands which have been executed thus far in the current plan session. This separates these values into equivalence classes where their real-life referent is the same. Of course this doesn't handle cases where a later `cd` comes back to a previous directory.

C.2 The Action Schemas

There were 43 valid command types used in the Linux corpus which we converted into the action schemas listed below. Each action schema lists the name of the command as well as its named parameters.

- `cal()`
- `cat(prepath,path)`
- `cd(prepath,path)`
- `clear()`
- `compress(prepath,path)`
- `cp(dest-prepath,dest-path,source-prepath,source-path)`
- `date()`
- `df(prepath,path)`
- `dir(prepath,path)`
- `du(prepath,path)`
- `echo(string)`
- `egrep(pattern,prepath,path)`
- `fgrep(pattern,prepath,path)`
- `file(prepath,path)`
- `find(prename,name,size,prepath,path)`
- `grep(pattern,prepath,path)`
- `gtar(dest-prepath,dest-path,source-prepath,source-path)`
- `gzip(prepath,path)`
- `info(command)`
- `jobs()`
- `less(prepath,path)`

- ln(dest-prepath,dest-path,source-prepath,source-path)
- ls(prepath,path)
- man(command)
- mkdir(prepath,path)
- more(prepath,path)
- mount()
- mv(dest-prepath,dest-path,source-prepath,source-path)
- pico(prepath,path)
- ping(machine-name,machine-path)
- pwd()
- rlogin(machine)
- rm(prepath,path)
- rsh(machine,command)
- ruptime()
- sort(prepath,path)
- tar(dest-prepath,dest-path,source-prepath,source-path)
- touch(prepath,path)
- tree(prepath,path)
- uncompress(prepath,path)
- vi(prepath,path)
- which(command)
- zip(dest-prepath,dest-path,source-prepath,source-path)

D Goal Schemas in the Monroe Corpus

There were 10 top-level goal schemas used in the Monroe corpus. Table D.1 shows each schema, along with its assigned a priori probability.

Note that, in the domain, we modeled roads as simple being between their two endpoints. Thus in `clear-road-hazard(from,to)`, the `from` and `to` parameters refer to the road between the two variables (where the hazard is).

Goal Schema	**Prob.**
`clear-road-hazard(from,to)`	0.094
`clear-road-tree(from,to)`	0.063
`clear-road-wreck(from,to)`	0.156
`fix-power-line(location)`	0.063
`fix-water-main(from,to)`	0.031
`plow-road(from,to)`	0.219
`provide-medical-attention(person)`	0.219
`provide-temp-heat(person)`	0.094
`quell-riot(location)`	0.031
`set-up-shelter(location)`	0.031

Table D.1: Goal Schemas in the Monroe Corpus