# GENERATING EFFECTIVE NATURAL LANGUAGE INSTRUCTIONS BASED ON AGENT EXPERTISE

Juliet C. Bourne

A DISSERTATION

 $\operatorname{in}$ 

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

1999

Dr. Bonnie Webber Supervisor of Dissertation

Dr. Jean Gallier Graduate Group Chair COPYRIGHT Juliet C. Bourne 1999 To all who have taught me, formally and informally, in my first thirty years.

## Acknowledgements

First and foremost, I must thank my advisor, Dr. Bonnie Webber, for all of her guidance and patience in my dissertation quest. The idea to use SPUD to generate instructions for my dissertation research came from her. In the final stages of producing this dissertation, she proved invaluable in keeping me focused and working on the important issues. She will always have my gratitude for all of her support in the last six years.

Secondly, I must thank Matthew Stone for writing SPUD and being a big help in general to me over the last several years. Without his work in developing SPUD, this dissertation, in its current form, would not have been possible.

Next, I must thank my dissertation committee members (Dr. James Allen, Dr. Norm Badler, Dr. Aravind Joshi, and Dr. Martha Palmer) for overseeing the completion of my dissertation. Their recommendation to consider the effect of agent expertise on the generation of instructions has obviously shaped the final form of my dissertation work.

I would like to thank everyone in the various groups I have participated in (DTOG, TOG/AMI, PAR, SPUD) for their helpful discussions out of which this dissertation began to take shape. In particular, I would like to thank Ed Boyle of HRGA at Wright-Patterson Air Force Base for his enthusiasm and support for research in generating instructions. He helped to shape the research that I pursued with the DTOG and TOG/AMI groups.

Many thanks as well to my professors at the University of Pennsylvania, especially Dr. Mark Steedman, Dr. Norm Badler, and Dr. Martha Palmer, for all of their help and encouragement throughout my graduate career. I would also like to thank Mike Felker for all of his hard work on my behalf, from beginning to end.

Finally, a big Thank You to my family, for their love and support.

This work has been supported by a Patricia Harris Fellowship, the U.S. Air Force through Delivery Orders #8 and #17 on F41624-97-D-5002, and the National Science Foundation through Grant #IRI95-04372.

## Abstract

#### GENERATING EFFECTIVE NATURAL LANGUAGE INSTRUCTIONS BASED ON AGENT EXPERTISE

Juliet C. Bourne Supervisor: Dr. Bonnie Webber

The automatic production of Natural Language instructions, i.e. those suitable for use by humans, has become an active area of research in recent years. In order for computergenerated instructions to be useful, they must be effective in accurately conveying the actions that are to be carried out by an agent. Conveying termination information, or when to stop performing an action, has been the focus of my dissertation research as it is an important part of generating effective instructions. I have done a corpus analysis of 3000 simple step-by-step maintenance instructions to study how termination information is expressed in naturally occurring texts. Using insights gained from the corpus analysis as well as the simulation of virtual agents carrying out similar tasks, I have specified an action representation, rules for reasoning about action information (particularly termination information), and some differences between agents with different levels of expertise. SPUD, a Natural Language generator developed at the University of Pennsylvania, takes this information, as well as information about the syntactic constructions used in instructions, and reasons about the best way to convey action information in effective instructions for particular agents.

## Contents

Acknowledgements			
A	bstra	let	vi
1	Intr	oduction	1
	1.1	Expressing Action Termination	3
	1.2	Representing Action Information	10
	1.3	Generating Effective Instructions	11
	1.4	Contributions and Overview of Work	12
<b>2</b>	Bac	kground	15
	2.1	Action Ontology	15
	2.2	Action Representation	19
	2.3	Modal First-Order Logic	20
	2.4	Lexical Choice	21
	2.5	Expressions of purpose	24
	2.6	Lexicalized Tree-Adjoining Grammar	26
	2.7	Natural Language Generation	31
3	Exp	pressing Action Termination	34
	3.1	About the Corpora	35
	3.2	Methodology	36
	3.3	Results and Analysis	41
		$3.3.1$ The use of verb arguments, particles, adverbs, and free adjuncts $\ldots$	43

		3.3.2	The use of prepositional phrases	44
		3.3.3	The use of purpose clauses	44
		3.3.4	The use of <i>until</i> clauses	47
	3.4	Concl	usion	49
4	Rep	oresent	ting Objects, Actions, and Agent Expertise	50
	4.1	The E	Example Domain	51
	4.2	Objec	t Information	52
	4.3	Action	n Information	55
		4.3.1	Parameterized Action Representation (PAR)	56
		4.3.2	Reasoning about action termination	61
		4.3.3	Reasoning about concreteness of actions	62
	4.4	Agent	Expertise Information	63
		4.4.1	Agent Types	64
		4.4.2	General knowledge	65
		4.4.3	Specific knowledge	66
		4.4.4	Reasoning about agent knowledge	68
	4.5	Concl	usion	71
5	Gei	neratin	ng Instructions using SPUD	73
	5.1	Basic	Lexical Information	74
		5.1.1	Specifying lexical items	74
		5.1.2	Specifying syntax	76
		5.1.3	Discourse and sentence structure	78
		5.1.4	Verb phrases	79
		5.1.5	Noun phrases	81
	5.2	Multi	-clausal lexical items	84
		5.2.1	Stylistic preferences	84
		5.2.2	Purpose and means clauses	86
		5.2.3	Until clauses	88
		5.2.4	Conjoining clauses	90

		6.2.3	Bateman and Paris, 1989	. 134
		6.2.3	Bateman and Paris, 1989	. 134
		6.2.2	Mellish and Evans, 1989	. 133
		6.2.1	Paris, 1988	. 132
	6.2	Other	approaches in Natural Language Generation	. 132
		6.1.6	Ghost Writer, 1996; <i>rating:</i> [-,-,-,~,?]	. 132
		6.1.5	IMAGENE, 1995; <i>rating</i> : [~(~), -, ~, -, ?]	. 131
		6.1.4	IDAS, 1995; <i>rating</i> : [~,+(+),~,-,?]	. 130
		6.1.3	TECHDOC. 1994; <i>rating</i> : [~,~,+,+,-]	. 130
		6.1.2	EPICURE, 1992; <i>rating</i> : [~,~,-,+,+]	. 129
	0.1	6.1.1	COMET. 1990: $rating:[-,-,~,+,+]$	. 129
	6.1	Instru	ction Generation Systems	. 128
6	Rel	ated W	Vork	128
	5.5	Discus	ssion	. 123
		5.4.4	Starting the pump	. 115
		5.4.3	Normalizing the pressure	. 109
		5.4.2	Opening the panel	. 96
		5.4.1	Generation instances	. 95
	5.4	Gener	ating Example Instructions	. 93
	5.3	The S	PUD Algorithm	. 91
		5.2.5	Multi-sentence instruction steps	. 90

$\mathbf{A}$	SPUD files 15		
	A.1	Lexical items	152
	A.2	Trees	163
	A.3	Modal operators	170
	A.4	Domain knowledge	170
	A.5	Morphological knowledge	181
	A.6	Generation instances	183
в	Exc	erpts from coded corpus	186
	B.1	$[Reader's Digest, 1991] \dots \dots$	186
	B.2	[USAF, 1988]	192
	B.3	[ITL SIMA, 1997]	194
$\mathbf{C}$	PA	R translation for SPUD	196

## List of Tables

3.1	Summary of codes used in corpus analysis	41
3.2	Overall Frequency of Main Clause Verb Phrase Codes	42
3.3	Frequency of Additional Codes	42
3.4	Code frequencies by corpora and main code	43
3.5	Co-occurrence of verb phrase types with purpose clauses	45
3.6	Purpose clause distribution by lexical item and main verb phrase code	45

## List of Figures

2.1	Mourelatos' typology of situations	16
2.2	Moens and Steedman's classification of events along two dimensions $\ . \ . \ .$	16
2.3	Moens and Steedman's tripartite structure of events/actions	17
2.4	Allen's thirteen relations between intervals	18
2.5	Possible locations for lexical choice in a NLG system	23
2.6	Tree for a simple sentence, $simpleS(S, R, E, A)$	27
2.7	Tree for transitive verbs, $transitive VP(S, R, E, A, O)$	27
2.8	Auxiliary tree $b VPpp(S, R, E, P, O)$	28
2.9	Example lexical items	28
2.10	Construction of example sentence	29
2.11	Construction of example sentence (continued)	30
3.1	Examples of $by$ purpose (means) clauses $\ldots \ldots \ldots \ldots \ldots \ldots \ldots$	46
3.2	Examples of (non-fronted) to purpose clauses	47
3.3	Examples of fronted to purpose clauses	48
4.1	A control panel for a pump	51
4.2	Portion of action/sub-action tree in the control panel domain $\ldots$ $\ldots$ $\ldots$	52
4.3	Knowledge about the screw	53
4.4	Part/whole object tree in the control panel domain $\ldots \ldots \ldots \ldots \ldots$	53
4.5	Object information for the lever's "RESET" position	55
4.6	The structure of PAR	57
4.7	PAR example and its translation into mFOL	60

4.8	Rules for time subintervals	61
4.9	Rules for reasoning about termination information	62
4.10	Rules for concreteness	63
4.11	Nested modal operators for agent types	65
4.12	Agent knowledge about opening the panel	67
4.13	Action information for opening the panel (and its sub-actions) $\ldots$ .	69
4.14	Action information for opening the panel (cont'd)	70
5.1	Lexical item entry for the verb <i>turn</i>	75
5.2	Full SPUD specification of the simpleS tree	77
5.3	Graphical specification of the simpleS tree	78
5.4	Basic discourse segment entry	79
5.5	Simple sentence entry	80
5.6	Predicative sentence entry	80
5.7	Transitive verb entries for open, remove, and turn	82
5.8	to lexical item (prepositional phrase) $\ldots$ $\ldots$ $\ldots$ $\ldots$	83
5.9	Lexical entry for an empty subject	83
5.10	Means clause entry	86
5.11	Discourse segment entry for emphasizing actions done for a purpose $\ldots$ .	87
5.12	Purpose clause entry	88
5.13	Fronted purpose clause entry	89
5.14	Until clause entry	89
5.15	Entry for conjoining verb phrases ("and then")	90
5.16	Multi-sentence discourse segment entry	91
5.17	SPUD's algorithm	92
5.18	Actions for example instructions	94
5.19	An example generation instance (Step 1 for Advanced) $\ldots$ $\ldots$ $\ldots$	95
5.20	Action information for opening the panel	97
5.21	Action information for removing the screw and the panel	98
5.22	Action information for how to remove the screw	99
5.23	Agent knowledge for Step 1	.00

5.24	Generation of Step 1 for Advanced, part 1
5.25	Generation of Step 1 for Advanced, part 2
5.26	Output: Step 1 for Advanced
5.27	Generation instance: Step 1 for Beginner
5.28	Generation of Step 1 for Beginner, part 1
5.29	Generation of Step 1 for Beginner, part 2
5.30	Output: Step 1 for Beginner
5.31	Generation instance: Step 1 for Novice
5.32	Generation of Step 1 for Novice, part 1
5.33	Generation of Step 1 for Novice, part 2
5.34	Output: Step 1 for Novice
5.35	Action information for normalizing the pump's pressure
5.36	Agent knowledge for Step 2
5.37	Output: Step 2 for Advanced
5.38	Lexical entry for <i>normalize</i>
5.39	Generation instance: Step 2 for Beginner
5.40	Output: Step 2 for Beginner
5.41	Lexical entry for <i>clockwise</i>
5.42	Output: Step 2 for Novice
5.43	Action information for starting the pump
5.44	Action information for resetting the pump
5.45	Agent knowledge for Step 3
5.46	Verb entries for Step 3
5.47	Output: Step 3 for Advanced
5.48	Generation instances: Step 3 for Beginner and Novice
5.49	Output: Step 3 for Beginner
5.50	Output: Step 3 for Novice

### Chapter 1

## Introduction

The automatic production of Natural Language instructions, i.e. those suitable for use by humans, has become an active area of research in recent years. The costliness of producing instruction manuals for complex systems, not to mention keeping the manuals updated, motivates the exploration into automating the process. On-line and interactive instruction systems are also gaining popularity for their ease-of-use and cost effectiveness. For any method of generating instructions to be successful, it must pay attention to the quality of the instructions that it produces or else what it produces will be useless for the purpose of replacing human-generated instructions. [Dixon, 1987] notes that when reading instructions, the reader must construct a mental representation (a plan) which is adequate for performing the given task. The constructed plan is only appropriate if it allows the reader to perform the task correctly and efficiently. Therefore, generated instructions must be *effective* in conveying the actions that are intended to be carried out so that the reader can form the appropriate plan. *Effective instructions* are those that convey all of the necessary information about performing the intended actions; if one piece of necessary information is missing, an instruction leaves open the possibility that an incorrect action will be performed. Therefore, merely automating the generation of instructions is not enough. Generating *effective* instructions should be at the heart of any system attempting to replace humans in producing instruction manuals and interactive instruction systems.

One aspect of Natural Language Generation is the description of entities so that they are distinguished from all other entities. Entities include objects, events, actions, and states of the world. Much attention has been paid to objects and the generation of their referring expressions, descriptions meant to pick out or refer to an entity. Creating a referring expression involves iteratively including properties which distinguish an object from the greatest number of its remaining distractors (other objects that it could be confused with) until the description can refer only to the intended object. For instance, if there are two blocks, one red and one blue, the referring expression for either of the blocks has to include its color property to distinguish it from the other block. The problem of generating referring expressions for entities other than objects has not been explored as thoroughly. An important part of generating instructions is distinguishing the actions that are to be carried out from similar actions. The generation of referring expressions for actions entails representing action information, distinguishing one action from another, and having a generation system that can consider information provided at the clause level (i.e. in a whole sentence) and can use multiple clauses (i.e. sentences with subordinate clauses) to describe actions.

Generating a distinguishing referring expression for an action is important, however it does not fully address the problem of producing effective instructions. Beyond distinguishing an action from other actions, effective instructions need to provide information that supports the performance of the action. Research involving the simulation of agents performing tasks helps define necessary action information which needs to be represented. The information that must be provided in order to have a simulated human agent carry out an action correlates, at some level, with the information that must be provided in Natural Language instructions to a human. When to stop performing an action represents a vital piece of action information. Without this *termination information*, an agent (simulated or otherwise) does not know when to stop performing an ongoing action. The action representation must therefore support termination information so that this information information, such as a post-condition or a path. Reasoning about action information to determine the implied termination information is therefore an important part of ensuring an action's performability.

The need for such reasoning requires that the system used for implementing the action

representation and generating instruction be powerful enough to support the formalization and use of rules for reasoning. Given the common need for action performance, a single representation of action information suitable for both simulated agents and Natural Language should be feasible. Such a representation would be independently-motivated, in that it would not be useful only for Natural Language purposes. It would represent all aspects of actions needed for performance and therefore it would support the generation of effective instructions. The Natural Language generator SPUD, developed at the University of Pennsylvania [Stone, 1998], has the representational and reasoning power needed for generating effective instructions. SPUD's method of generation allows it to flexibly produce descriptions of actions that are linguistically-sound, not *ad hoc*. Using SPUD's ability to reason about the information conveyed by action descriptions, even partial ones, generating effective instructions is reduced to specifying rules for reasoning about action information and encoding the ways of conveying action information.

In the rest of this chapter, some introductory details are presented regarding the main areas of this dissertation. First, expressions of action termination are introduced in more depth. Next, action and agent representations and their relation to effective instructions are discussed. Finally, the method of generation is briefly outlined. This chapter ends with the claim of this dissertation, an outline of the contributions of this dissertation work, and an overview of the rest of the dissertation.

#### **1.1 Expressing Action Termination**

As alluded to earlier, an important goal of generating Natural Language instructions is to describe actions fully and accurately so that they can be carried out correctly. This goal is particularly important to the generation of written instructions where the "speaker" (i.e. author) and the "hearer" (i.e. reader) are separated spatially and temporally. In the case of instruction manuals, the hearer does not have the opportunity to ask questions to clarify the action to be performed and the speaker likewise does not get any feedback from the hearer about the success of the instructions. Therefore, attention must be paid to the *effectiveness* of the instructions generated to be sure that they can be carried out correctly.

Attention must also be paid to the efficiency or conciseness of the instructions. That is, all the necessary information should be included in an efficient manner in order to avoid confusion caused by extra information. Understanding how information about an action is expressed, which ways of expressing information are used for which purposes, etc., is essential to generating instructions that describe actions both effectively and efficiently.

How is action termination expressed in instructions? The answers have been found through the analysis of naturally-occurring texts in terms of linguistic constructions (i.e. ways of expressing information in Natural Language) used to describe actions. Linguistic constructions include required verb arguments (e.g. "Rotate the dial"), optional verb adjuncts (e.g. "Rotate the dial 90 degrees"), prepositional phrases (e.g. "Rotate the dial to the 90-degree mark"), until clauses (e.g. "Turn screw until it is loose"), etc. All are used to describe necessary pieces of information about actions; in fact, all of the linguistic constructions mentioned above can be used to provide termination information. If termination is missing, then an instruction may be ineffective unless it is known that the hearer can infer the correct termination information from the action information persented. Termination information can be conveyed both through explicit termination information instance. Therefore, knowing how action information can be expressed is necessary to generate effective instructions.

Expressing action information has been explored in several ways by other researchers. For instance, the issue of *lexical choice*, choosing the words to describe an entity, has been addressed by a number of researchers (e.g. [Reiter, 1991; Kosseim and Lapalme, 1995; Elhadad *et al.*, 1997]). The generation of *referring expressions* for objects at various points in a set of instructions, has also been explored [Dale, 1992]. Expressing the *purpose* of an action, i.e. "Do x to do y," has been examined by several researchers as well [Di Eugenio, 1993; Vander Linden, 1994; Kosseim and Lapalme, 1995; Vander Linden and Martin, 1995; Di Eugenio and Webber, 1996; Hartley and Paris, 1996]. Despite all of this previous work, discussed in Chapter 2, the issue of expressing an action's termination remains unexplored.

Before discussing how termination information is expressed, I should clarify the terms

which I will be using to refer to actions and their descriptions in Natural Language:

- Action class refers to a set of actions with the same main semantic components (e.g. motion of an object). However, sometimes I will also use this to refer to more specific action classes incorporating certain types of objects. The context in which the term is used should distinguish whether I am referring to a general or specific action class, if it makes a difference.
- Action instance (or just action) refers to a particular action in the world, complete with specific properties and particular entities.
- Action description refers to the set of linguistic expressions used to describe a particular action instance. These expressions do not necessarily have to be contiguous in the actual text and they can appear across multiple sentences.
- **Instruction** refers to a single sentence with an imperative main clause which describes an action (or actions) for the hearer to perform.<sup>1</sup> It need not contain a complete action description.
- **Instruction step** refers to a set of instructions describing a single step of a task. The task step could involve multiple actions, and therefore could require multiple instructions.

Actions have different types of aspectual (temporal) structure (discussed in Chapter 2) and the type of an action can provide termination information. For instance, *culmination* (which is termination plus a change of state) can be inherent in some actions, such as *removing* and *breaking*. For these actions, just giving the main component of the action, i.e. the change of state, provides the termination information. However, some actions, such as *turning*, do not have inherent culmination or termination. These actions, called *activities*, need to have termination information included in their action descriptions in order to produce effective instructions. Termination information can be explicit in the instructions, as mentioned before, or implicit in the interaction of the activity with other actions in the instruction step.

<sup>&</sup>lt;sup>1</sup>This restrictive definition is used because (as will be discussed later) the corpus analysis done for this dissertation work is restricted to such instructions. Non-imperative instructions, such as modal instructions (e.g. "You should do X"), do not appear in the subset of maintenance instructions used to build the language model.

In Natural Language, information about an action is *realized*, or expressed in Natural Language by many different linguistic sources. For example, the main component of the action is usually expressed through the verb. Action verbs reflect the different aspectual types of actions. For instance, the verb *remove* is considered an *accomplishment* verb, which means that it has inherent culmination (among other things). However, the type (and thus termination) of an action is determined by *all* of its information. Thus, linguistic expressions for other parts of the action, such as the arguments to the verb and additional phrases such as purpose clauses and temporal clauses, can change the type of the action [Moens, 1987]. Interactions among these linguistic expressions affect the type of the action expressed and should be considered when deciding how to describe an action.

The variety of linguistic constructions which express termination information provides several choices for expressing the termination of an action, each with different implications in different contexts. Characterizing the choices made in naturally-occurring instructions (e.g. through a corpus analysis) determines how to automate the same choices in order to produce natural and fluent instructions. Below I describe the characteristics of the corpus instructions examined in terms of expressing action termination. These characteristics demonstrate the genre of instructions generated by the implementation discussed in this dissertation.

#### Corpus domain

The corpus analysis examines a corpus of simple step-by-step maintenance instructions which includes parts of a "do-it-yourself" book for home maintenance and a collection of *technical orders* (military instructions) for the maintenance of F-16 aircraft. It considers only the numbered step-by-step parts of the texts rather than the general discussion in the former and the notes, cautions, and warnings in the latter. In a corpus study done by [Hartley and Paris, 1996], such step-by-step instructions are shown to be a sub-genre of instructions. Their analysis shows that step-by-step instructions have linguistic features, such as the dominance of imperative sentences, which distinguish them from other subgenres (e.g. reference and tutorial texts). Thus, restricting the corpus to step-by-step instructions is linguistically-motivated and provides a manageable collection of contexts and linguistic features to study.

Since the domains covered by the corpus of instructions are complex, the actions described by the instructions are also complex and varied, including those involving motion over time. Having such actions, as opposed to "change of state" actions, means that some actions do not have inherent culmination and thus need termination information in their descriptions. As I show in Chapter 3, termination information is usually explicit in each instruction<sup>2</sup>, either because the action has an inherent termination or because an expression giving or implying termination is included.

#### Constructions

As noted above, termination information has many sources in an action description. These sources fall into the following groups:

- **Predicate-argument structure** consists of the verb and its required arguments, denoting the participants of the action and other essential information. The verb alone can have an inherent termination, as in
  - (1) Remove the access panel.

or the verb combined with its specific argument or type of argument can give a termination, as in

- (2) Cut the wire.
- (3) Pour one cup of water into the bowl.
- Additional phrases, e.g. prepositional phrases (paths or locations), adverbial phrases (direction and manner), etc., can also give termination information. For example:
  - (4) Rotate aerial refueling control to full counterclockwise (off) position.[USAF, 1988]

 $<sup>^{2}</sup>$ In the complex portion of the corpus (i.e. the "do-it-yourself" instructions), termination is sometimes left to be inferred from knowledge the hearer is assumed to have about the domain.

The action description without the prepositional phrase, i.e. "*Rotate the aerial refueling control*", does not express when to stop the action of rotating the control. Another example:

(5) Clamp work securely, and mark positions of screws. [Reader's Digest, 1991]

In this example, the adverb *securely* indicates not only the manner of the clamping but also when the action of clamping can stop (i.e. when the work is secure).

- Additional clauses, such as *until* and *when* clauses, purpose clauses (including purposive and clauses), etc., can provide the termination of an action. The following examples are from the corpus of F-16 maintenance instructions [USAF, 1988]:
  - (6) a. Depress system A reservoir dump valve until accumulator gage [sic] indicates precharge pressure.
    - b. *Slide valve aft* and remove.
    - c. Depress bleed valve sufficiently to obtain stream of fluid flow.
- Interaction between an action and other actions, i.e. whether a generation or enablement relationship exists between two actions, whether one action is done for the purpose of another, whether the start of the next action implies the termination of the previous one, etc., can give the termination of an action. Such non-lexicalized sources of termination information require inference on the part of the hearer. For instance:
  - (7) Hold bit against fence and wheel; roll it clockwise and swing it to 12 degree line. [Reader's Digest, 1991]

In this example, the actions of holding and rolling the bit acquire their terminations from the last action in the sequence, that of swinging it to the 12-degree line. They are performed until the last action is completed. No lexical information explicitly indicates the first two actions' terminations.

For any action, termination information can be combined from multiple sources as seen in this example [USAF, 1988]:

(8) NOTE: To remove actuator, it will be necessary to lift actuator slightly and rotate actuator 90 degrees clockwise until sufficient clearance is obtained to disengage actuator splines.

This example is not in the step-by-step corpus. It is shown as a good example of multiple sources of termination information and could be paraphrased as a step-by-step instruction as

(9) To remove actuator, lift actuator slightly and rotate actuator 90 degrees clockwise until sufficient clearance is obtained to disengage actuator splines.

showing the complex interaction of action information in order to provide termination information.

While the corpus analysis shows that predicate-argument structure commonly conveys termination, the way in which it provides termination information is semantically complex. Similarly, gaining termination information through interactions with other actions is even more complex, but it is not as frequent. The other two groups of sources account for a significant portion of the corpus in which termination information is gained explicitly while being semantically simpler. Thus, limiting the implementation of termination expressions to these two groups does not greatly restrict the coverage of the naturally occurring data and allows simple characterizations to be used.

Using a particular construction in an instruction causes the hearer to make some assumptions about the world and the action to be performed. Characterizations of certain constructions gleaned from the corpus study are discussed in Chapter 3 and encoded as shown in Chapter 5. A related question is that, when termination information for an action is not explicit in the instructions, is an *expectation* raised that the termination is assumed to be known, inferable, or otherwise defaulted to by the hearer? The implementation of domain knowledge, given in Chapter 4, discusses this point with respect to what the hearer is assumed to know about the domain and its actions. In this way, the hearer's ability to infer termination information from an action description can be reasoned about and can guide the generation process.

#### **1.2 Representing Action Information**

The Parameterized Action Representation (PAR), developed at the University of Pennsylvania, is meant as an intermediate representation that can support both the animation and Natural Language description of actions [Badler et al, 1997; Badler et al, 1998; Badler et al., 1998]. As such a representation, PAR can represent actions at various levels of abstraction, from general action classes to specific action instances (i.e. sets of action performances). A PAR instance for an action consists of the features of the action, including the main semantic components of the action which identify its general action class. Other features include specific information about the action which distinguishes it from others in its action class. For the purposes of providing information relevant to the termination of an action as described above, the PAR needs to represent the following pieces of information:

- core semantics the state-change, motion, and/or forces of the action
- direction/path the direction or path of motion or force
- **purpose** the purpose for which the action is done: to achieve a particular state, to generate another action, and/or to enable the next action
- termination explicit termination conditions (events or states of the world) unrelated to other aspects of the action
- duration explicit timing of the action (e.g. for 6 seconds) or iteration (e.g. between 5 and 6 times)

Each of these can be realized in an action description. For instance, the core semantics is usually realized as the verb, the path as a prepositional phrase, etc.

For actions that are part of an instruction step or that contain sub-steps of their own, the PAR also needs to include information about the other actions in the instruction step. For example, if an action has sub-steps, these should be given in the PAR instance for the action. Likewise, a sub-step action instance should have pointers to the action instance which it is a sub-step of as well as the other sub-steps.<sup>3</sup> Therefore, the following pieces of

<sup>&</sup>lt;sup>3</sup>Although only specific action instances are considered for the implementation, these pieces of information could also be used when specifying the properties of action classes or generic actions.

information, each of which can also have a linguistic counterpart, are included in the PAR:

- subactions elaboration of how to accomplish the action
- previous action link to a previous action
- concurrent action link to a concurrent action
- next action link to a following action
- parent action link back to the parent action of which the action is a sub-step

The structure of the PAR is discussed in detail in Chapter 4. Rules for reasoning about action information, e.g. determining whether termination information is implied, are also presented. In addition, rules defining what particular agents know about actions in the domain are discussed. These rules for reasoning about actions and agents both contribute to generation of effective instructions.

#### **1.3 Generating Effective Instructions**

A Natural Language generator called SPUD (Sentence Planning Using Descriptions) [Stone and Doran, 1997; Stone and Webber, 1998; Stone, 1998], developed at the University of Pennsylvania, provides the necessary components for generating effective instructions. SPUD forms descriptions of actions (as well as events, states, and objects) by choosing lexical items from its Lexicalized Tree-Adjoining Grammar (i.e. words and their associated linguistic constructions) which serve the given communicative goals best. Lexical items are annotated with semantic and pragmatic information that SPUD can match against the information that it is trying to convey. Using this framework to implement multiclausal sentences and multi-sentence discourse, instructions are generated from the action representation and the semantic and pragmatic contexts determined empirically in the corpus analysis.

Encoding constructions for SPUD consists of creating lexical items which specify their syntax as well as the semantic and pragmatic contexts in which they are used (described in Chapter 3). Additional communicative goals are given to SPUD as part of a generation task in order to ensure that the necessary action information is conveyed. To determine if a particular action description is effective, SPUD uses rules that it can use to check whether the description provides termination information and other necessary action information; these are presented in Chapter 4. Given all of this information, SPUD can be told to generate a description of a particular action in the form of an instruction for a particular agent. SPUD uses the semantic and pragmatic context to determine the best description of the action, making sure that it satisfies the communicative goals (which in turn ensure that it is effective for the performance of the action). SPUD, the encoded constructions, and the generation of example instructions are described in more detail in Chapter 5.

#### 1.4 Contributions and Overview of Work

In this dissertation, I claim that the generation of effective instructions can be accomplished through determining and encoding how action information is expressed naturally, representing the necessary action information and formalizing rules for reasoning about it, modeling agent knowledge, and providing all of this information to a Natural Language generator capable of considering and reasoning about the information in an action description and its effect on an agent's knowledge and ability to perform the action. In order to support this claim, I have three interrelated goals in my dissertation work: characterizing particular constructions used for expressing action termination, representing actions and domain knowledge, and generating instructions from action instances and agent knowledge. The contributions of this dissertation in these areas include:

- analysis of action types and termination information in naturally occurring instructional texts;
- use of an action representation developed for and capable of representing necessary action information for performing actions;
- implementation of rules for reasoning about the effectiveness of action descriptions, with respect to agent knowledge in the areas of concreteness and termination information (i.e. action performability);

- implementation of agent expertise levels using modal operators for use in the generation of tailored instructions;
- demonstration of the generation of different instructions from the same action information based on the assumed knowledge of the agents.

In Chapter 3, constructions used for expressing termination, along with the semantic and pragmatic contexts in which they appear, are analyzed and characterized based on actual language use. I have coded a corpus of instructions for the types of actions that occur, the constructions which appear in the action descriptions, and the sources of termination information. From the coding, I have analyzed the instructions and drawn conclusions about the use of certain constructions for expressing termination information. The characterization of the constructions are later expressed in the domain knowledge and linguistic representation for the Natural Language generator.

In Chapter 4, I detail the implementation of domain knowledge, which includes representing action information as well as agent knowledge. The action representation supports the necessary information about actions, including termination information, and includes the relationships between actions, such as sub-steps and purposes, which can be sources of termination information. I argue that the representation should be as *language-neutral* as possible; that is, it should not be structured in a certain way just for linguistic reasons. The action representation must be suitable for generating Natural Language from and yet not be tied to any particular language or linguistic theory. The same holds true for the rest of the domain knowledge, including knowledge about objects and agents. In addition, I formalize the reasoning about the performability of actions and the experience that agents have.

In Chapter 5, I bring the previous two chapters' work together in the generation of example instructions in the domain. The constructions for expressing action information are encoded for SPUD so that it can create action descriptions based on the same semantic and pragmatic contexts determined in the corpus analysis. The encoded constructions give SPUD the option of spreading an action description over multiple clauses and sentences, since information about an action (especially termination information) can appear in multiple clauses and sentences. Stylistic preferences for different agent types are used to guide SPUD in such choices. Using knowledge about the domain and the agent and reasoning about the information provided by constructions, SPUD determines the best way to express action information for a particular agent and the given communicative goals.

In Chapter 6, I discuss related generation systems and compare them with the implementation presented in this dissertation. In Chapter 7, I conclude this dissertation with a summary of what has been accomplished and what is left as further work. In the next chapter, I discuss background material in the areas of action ontology, domain knowledge representation, lexical choice and linguistic constructions for termination information, and Natural Language Generation.

### Chapter 2

## Background

The following sections present background material for topics mentioned in this dissertation. The first, and most central, is that of the classification of action types, presented in the first section. Next, the representation of actions is discussed. Then modal firstorder logic is presented, as it is used to represent domain knowledge (objects, actions, and agents). After this, the remaining sections of the chapter address various aspects of Natural Language Generation (NLG) research: lexical choice, or how words are chosen; work on the expression of purpose in actions; LTAG, a grammar formalism; and finally, the major issues for NLG systems. This material is meant to introduce some terms and concepts referred to and relied upon later. Comparisons between the implementation discussed in this dissertation and other NLG systems are presented in Chapter 6.

#### 2.1 Action Ontology

As far back as Aristotle, philosophers and linguists have pondered the types of situations (events, actions, and states) evoked in language. Vendler [Vendler, 1967] proposed a typology of situations, distinguishing between *accomplishments*, *achievements*, *activities*, and *states*, each of which has its own temporal structure and properties. An activity, such as *pushing a cart*, has "no set terminal point," while an accomplishment, such as *drawing a circle*, has "a 'climax', which has to be reached if the action is to be what it is claimed to be" [Vendler, 1967, p.100]. Achievements, such as *reaching the top*, "occur at a single



Figure 2.1: Mourelatos' typology of situations

EVENTS				
	$\operatorname{atomic}$	extended		
+ conseq	Harry broke the window	Sue built a sandcastle		
(telic)	$(\mathbf{achievement})$	$(\mathbf{accomplishment})$		
-conseq	Sandra hiccupped	Max worked in the garden		
(atelic)	$(\mathbf{point})$	$(\mathbf{activity})$		

Figure 2.2: Moens and Steedman's classification of events along two dimensions

moment", whereas states, such as *loving*, "last for a period of time" [Vendler, 1967, p. 103]. Mourelatos [Mourelatos, 1981] proposed a similar typology, but he collapsed accomplishments and achievements together as *events* (see Figure 2.1, adapted from [Passonneau, 1987, Figure 1]). Moens and Steedman [Moens and Steedman, 1987] follow in the same vein, classifying situations into states and events. However, they make a finer and more systematic distinction between the kinds of events (and, therefore, actions). Events/actions are characterized along two dimensions — the extension of an event or action in time (or, alternatively, its ability to be decomposed into sub-events or sub-actions) and the existence of characteristic consequences associated with the event or action (see Figure 2.2, adapted from [Moens and Steedman, 1987, Figure 1]).

While all four types of events/actions shown in Figure 2.2 exist, actions which appear in maintenance instructions tend to be either achievements or accomplishments. Both



Figure 2.3: Moens and Steedman's tripartite structure of events/actions

of these types have consequences, or effects on the world, which is the general point in maintenance tasks. Another feature they have in common, related to the fact that they have consequences, is that they have defined endpoints. That is, achievements and accomplishments, as part of their meaning, include when to stop doing the actions. This inherent termination can be seen in the tripartite representation of actions that [Moens and Steedman, 1987] propose (Figure 2.3). In this representation, actions can have a preparatory process, a culmination point, and a consequent state. The culmination point, right before the consequent state begins, is the termination of both achievements and accomplishments. The difference between the two types is that an achievement does not have a characteristic preparatory process leading up to the culmination. Despite this, they are interchangeable by stripping away or adding the preparatory process, depending on the importance placed on the preparatory process.

An important part of understanding instructions is understanding how the different actions in an instruction step are related temporally (as well as causally). While instructions are usually given in the order in which they are to be done, it is still sometimes necessary to express more complex temporal relationships, such as overlap or concurrency. Allen [Allen, 1983; Allen, 1984] has identified a set of thirteen temporal relations between the intervals (spans of time) over which situations hold or take place, shown in Figure 2.4 (adapted from [How, 1993, Figure 2.5]). Three of these (*meets, starts, and finishes*) figure prominently in the representation of actions and their associated time intervals in this dissertation.



Figure 2.4: Allen's thirteen relations between intervals

#### 2.2 Action Representation

First-order logic, description logics, and feature structures have been used in representing actions for various purposes. Steedman [Steedman, 1997] has proposed encoding the semantics of events/actions in a dynamic semantics formalism, an extension of modal logic where the occurrences of actions defines accessibility relations between possible worlds. Dynamic semantics, as well as predecessors such as situation calculus and event calculus, is meant as a representation for reasoning about actions, their temporal structure, and their consequences. The generation system COMET [McKeown et al., 1990] uses Functional Unification Formalism, an extension of functional unification grammar (related to feature structures), to represent logical-form semantics of actions. COMET uses this representation for lexical choice since the representations of actions and the representations of linguistic constructions can be unified (i.e. their attributes compared and combined in certain ways) to form descriptions of actions. Description logic representations, i.e. combinations of feature structures and logic machinery, include CLASSIC (used by Di Eugenio, 1993, discussed briefly in Section 2.5) and LOOM (used by [Rösner and Stede, 1994], among others, as discussed in Chapter 6). Description logic representations have been used to reason about the properties of actions and how they can be classified in an action classification hierarchy. Description logics have even been used to represent linguistic knowledge, classifying the semantics of linguistic constructions. Overall, feature structures are the simplest and most common way of representing actions. Feature structures contain attribute-value pairs (e.g. [agent = you]) where the value is a simple token or another feature structure. [Dale, 1992] is one good example of using feature structures to represent action information, although the main focus is on object information (see Chapter 6). PAR, the action representation used in this dissertation, can be implemented as a feature structure representation and is discussed as such in Section 4.3.1. Feature structures form the basis of most action representations because of their simple attributevalue structure. While they suffice for representing information about actions, additional machinery is needed to reason about the actions represented. In the next section, modal first-order logic is discussed; it is the knowledge representation used in the implementation described in this dissertation.

#### 2.3 Modal First-Order Logic

First-order logic (FOL) consists of atoms, predicates, quantifiers, and variables combined into formulas using logical operations of conjunction (and), disjunction (or), negation, and implication. Formulas in FOL can express facts and rules about a domain. For example, the following are FOL formulas:

#### bird(tweety)

 $\forall x \quad (\operatorname{bird}(x) \to \operatorname{flies}(x))$ 

The first formula states the fact that the predicate **bird** holds for the atom **tweety**, a particular object in the domain. The second states the rule that for all objects (using the universal quantifier  $\forall$  and the variable x) for which **bird** holds, the predicate **flies** also holds. In English, these mean that "Tweety" is a bird and that all birds fly, respectively. Using logical theorems which determine how to manipulate logical statements, these formulas can be used to prove the formula **flies**(**tweety**), representing the reasoning that since "Tweety" is a bird, "Tweety" can fly. A straightforward correlation can be made between the *representational* power of feature structures and that of first-order logic. First-order logic can represent anything represented in a feature structure. By using FOL predicates corresponding to feature structure attributes and assigning identifiers to entities (objects, actions, etc.), the same information found in feature structures can be represented in FOL. For instance, the object feature structure for "Tweety" could be [**id = tweety; type = bird**] and translated as the first FOL formula presented above. Thus, FOL is suitable for representing and reasoning about many domains, particularly those involving objects and their properties.

While FOL is powerful and can be used to represent and reason about actions, it is not fully capable of representing and reasoning about the knowledge that *agents* might have in a domain. In order to do so, a representation needs the ability to state that certain pieces of knowledge are assumed to be known by certain agents. This cannot be done with standard FOL, as all statements of knowledge have the same status (e.g. known by everyone). One solution is to combine first-order logic and modal logic. Modal logics denote the informational status of pieces of knowledge by predicating them with *modal*  operators.<sup>1</sup> For instance, a modal operator, say  $\mathbf{U}$ , could be used to represent an agent's assumed knowledge, and another, say  $\mathbf{S}$ , could be used to represent the system's knowledge. Using these modal operators, what the agent is assumed to know can be reasoned about separately from what the system knows. The relationship between these two operators could be defined so that, in addition to other knowledge, the system knows everything the agent is assumed to know; treating operators as sets of knowledge, this relationship would be represented in set terms by  $\mathbf{S} \supset \mathbf{U}$ . For example, if the system knows that "Tweety" is a bird and both the system and the agent know the rule about all birds flying, the FOL formulas from before would be stated in modal FOL as

S bird(tweety) U  $\forall x(bird(x) \rightarrow flies(x))$ 

Using these formulas, the formula **S** flies(tweety) (i.e. the system knows that "Tweety" can fly) can be proven but **U** flies(tweety) cannot. The fact that "Tweety" is a bird is not predicated with the **U** modal operator and thus the agent does not know that "Tweety" is a bird and cannot infer that "Tweety" can fly. In this way, agent knowledge can be represented and reasoned about, distinct from other knowledge predicated about the domain.

#### 2.4 Lexical Choice

"The problem of determining what words to use for the concepts in the domain representation is termed lexical choice. In an effort to make domain representations independent of language, there may be a variety of different words that can be used to express any concept in the domain, and a language generator must choose which one is most appropriate in the current context." [Elhadad et al., 1997, p.195]

The choice of words and linguistic constructions (i.e. syntax) anchors the generation of instructions. Words and constructions need to be chosen based upon their meaning and

<sup>&</sup>lt;sup>1</sup>In the case of dynamic semantics, an action representation mentioned in the previous section, modal operators are used to represent the effects of actions, in the sense of representing the state of the world after an action occurs.

implications in expressing information. Lexical choice implementations rely on analyses of words and constructions in natural texts. The choice of a particular word or construction to express a piece of information depends on many contextual factors. Contextual factors include previous syntactic and lexical choices, since they can affect the choices that can be made subsequently. The structure of the domain, e.g. its objects and relations, also affects lexical choice as it may force or preclude particular choices. What is commonly thought of as "the context," that is, information about the speaker, the hearer, and the previous discourse, also contributes additional contextual factors. All of these contextual factors *constrain* the choice of lexical items and their syntactic constructions, as described clearly by [Elhadad *et al.*, 1997].

The development of a lexical choice algorithm begins with determining correlations between contextual factors and linguistic features of words and constructions, usually through a corpus analysis as demonstrated by [Hartley and Paris, 1996], among others. Once the contextual factors and the ways in which they constrain the range of linguistic features have been determined, several methods can be used to perform lexical choice. Since generation systems depend on lexical choice to determine the most appropriate way to express information, lexical choice algorithms represent one of the key aspects of generation systems. Lexical choice methods differ in a number of ways, including the constraints which they consider, how those constraints are represented, the location of lexical choice in the system architecture, and what the lexical choice algorithm receives as input. The constraints used by a system determine its ability to choose between similar words and constructions. If the constraints are general, then the lexical choice algorithm will be able to make only coarse-grained decisions. In addition, the representation of the constraints, e.g. as rules or heuristics, affects the location of lexical choice. If constraints are purely semantic (i.e. determined only by content), then lexical choice can be done with content planning (i.e. at location 1 in Figure 2.5). The advantage of this is that the lexical choice algorithm has access to the domain representation, however this may mean that concepts and words have a one-to-one correspondence, i.e. the same words are always chosen for the same concepts, reducing the expressive flexibility of the system. Another disadvantage is the fact that if it is discovered in surface realization (i.e. the choice of syntactic constructions)


- 1. Particular words to use are chosen during content planning.
- 2. Words are chosen after the complete content is specified.
- 3. Words are chosen simultaneously with choosing syntax.

Figure 2.5: Possible locations for lexical choice in a NLG system

that the chosen words cannot be used due to syntactic constraints, lexical choice must be redone. At the other extreme in the system architecture shown in Figure 2.5, where lexical choice is done with surface realization, constraints on lexical choice can rely on semantics as well as syntax. This avoids the problem of having to redo lexical choice because of syntactic constraints; however, depending on the particular architecture, lexical choice algorithms might be deprived of consulting the domain representation as part of the decision-making process. What input is provided to the lexical choice algorithm, i.e. the information on which it bases its decisions, determines the quality of the decisions made. Not enough information or the wrong kind of information can result in poor lexical choice. All of these factors determine how well a lexical choice algorithm is able to choose appropriate words or linguistic constructions.

# 2.5 Expressions of purpose

While the number of different linguistic constructions is considerable, those involving expressions of purpose have been the focus of much research, especially in terms of their use in instructions. Since the performance of an action can change depending on the purpose for which it is done, conveying an action's purpose is important in instructions. Purpose can modify many aspects of the performance of an action, including its termination and manner. The decision to include purpose constructions in this dissertation stems from their use to convey termination information for actions<sup>2</sup> and from the fact that understanding how to express purpose is necessary in general in order to produce natural and effective instructions. Thus, I briefly review some relevant research which explores how expressions of purpose are related to the semantics of actions.

- [Thompson, 1985] presents a corpus study in which she analyzes initial (fronted) and final (non-fronted) purpose clauses. She discovers that, rather than being a choice between putting a "purpose clause" before or after the main clause, initial and final purpose clauses act as very different constructions. In a corpus of narratives, procedural texts, and a Master of Arts thesis, she found that only 18% of the purpose clauses were initial purpose clauses and the rest were final purpose clauses. However, in the two procedural texts, the percentage of initial and final purpose clauses were significantly different: a 50%/50% split in one text and approximately a 25%/75% split in the other. The different roles that initial and final purpose clauses play cause this distinction between procedural texts and non-procedural texts. [Thompson, 1985] characterizes initial and final purpose clauses this way:
  - An initial purpose clause states a problem within the context of expectations and the following clauses (often many) provide the solution to the problem. In other words, initial purpose clauses guide the reader, providing a framework for interpreting the main clause which is typically "weighty".

 $<sup>^{2}</sup>$  As the next chapter shows, nearly a third of the purpose constructions in the corpus provide termination information.

A final purpose clause states the purpose for doing the action in the main clause.
Serving such a local role, the scope of the final purpose clause is restricted to the immediately preceding main clause (which gives an action by volitional agent).
Based on the non-use of commas (as opposed to initial purpose clauses), final purpose clauses are more tightly linked to the action in the main clause.

This distinction between initial and final purpose clauses accounts for the fact that procedural texts use more initial purpose clauses than other types of texts since they are organized in terms of problems and solutions and thus favor initial purpose clauses.

- [Di Eugenio, 1993; Di Eugenio and Webber, 1996] look at purpose clauses with respect to inferences that must be made to interpret instructions. They consider how actions are related as well as the assumptions made to accommodate such relations. While they deal with interpretation rather than generation, their analysis and conclusions are valuable and can be applied to generation.
- [Kosseim and Lapalme, 1995] develop heuristics for determining how to express effects and guidances. Effects are essentially generation relationships between actions and other actions or events. Guidances are conditional generation relationships between actions, i.e. the action to be generated will only occur if certain conditions hold. This work explores how to realize these "semantic carriers" (*rhetorical relations*) as purpose clauses, means ("by") clauses, or statements of result.
- [Vander Linden and Martin, 1995] perform a corpus analysis to determine correlations between contextual factors (e.g., semantics, discourse, and the hearer model) and the ways in which purpose is expressed in instructions. The decisions that are made about the purpose expression include: its slot (position with respect to main action), its form (grammatical category), its linker or cue words (fixed lexical items in constructions), and how clauses are combined.
- [Hartley and Paris, 1996] encode correlations of task elements and linguistic features in a strata of networks for realization (e.g. lexical and syntactic) choices. The task

elements include goals, functions, constraints, etc., in the domain of software instruction manuals. The realization choices are based on systemic functional linguistics (SFL) and they use a SFL-based tactical generator.

While the methods for choosing between forms of purpose expressions differ, one factor remains fairly constant across discussions of choosing purpose expressions: the use of corpus analyses. A corpus analysis, the study of naturally occurring texts, is the basis for the decisions which are encoded in lexical choice algorithms. To some extent, I have incorporated the previous work done on purpose constructions into the implementation in this dissertation, in terms of the types of purpose relationships between actions as well as the overall method for arriving at characterizations of particular constructions and how to choose between them. This is discussed is more detail in the next chapter, which presents the corpus analysis done for this dissertation.

# 2.6 Lexicalized Tree-Adjoining Grammar

Every NLG system needs some way to represent the grammar, or language model, for the texts it produces. In this implementation, the Natural Language generator, SPUD, uses a Lexicalized Tree-Adjoining Grammar (LTAG) [Schabes, 1990]. Tree-Adjoining Grammar (TAG) is a syntactic formalism in which trees define individual pieces of syntax and operations to combine trees denote how syntactic components can interact [Joshi *et al.*, 1975]. LTAG is a variant of TAG in which each syntactic tree is *anchored* by (i.e. associated with) at least one lexical item (word). For example, consider the trees shown in Figures 2.6 and 2.7, presented in SPUD's non-traditional fashion.<sup>3</sup> The first shows a tree with an inflectional item anchoring a sentence (S) which consists of a subject (NP) and a predicate (IP).<sup>4</sup> Unlike most other trees, lexical items which anchor this tree, i.e. tenses such as

<sup>&</sup>lt;sup>3</sup>This graphical tree format is used for readability purposes. The actual input format to SPUD is shown in Appendix A.2. Although not shown in the graphical format, syntactic features as well as pragmatic information are associated with each tree. Technically, the addition of features makes this *Feature-Based* LTAG. The addition of semantic indices (e.g. (S,R,E,A)) and pragmatic information makes this use of LTAG non-standard.

<sup>&</sup>lt;sup>4</sup>Treating inflection as anchors for sentence trees differs from most LTAG uses which usually use verbs as anchors for sentence trees. The decision to treat inflection this way does not have a bearing on the rest of the work in this dissertation and could have been done differently without significant change to the results.



Figure 2.6: Tree for a simple sentence, simple S(S, R, E, A)



Figure 2.7: Tree for transitive verbs, transitive VP(S, R, E, A, O)

present or past, are realized by affecting features of other parts of the tree (in this case, the inflectional form of the verb). The second tree (Figure 2.7) shows a transitive verb phrase which consists of a verb and its object. Each node is labeled with its syntactic category (e.g. V for the verb and NP for the object). In SPUD's use of LTAG, the semantic entities being described by the node are also indicated, e.g. NP(A) is a noun phrase describing the entity A.<sup>5</sup> The leaves with downward arrows ( $\downarrow$ ) indicate substitution sites, where trees of the right category can be inserted into the tree. The diamond ( $\diamond$ ) indicates the position of the lexical item which anchors the tree.

The types of tree shown in Figures 2.6 and 2.7 are called initial or *alpha* trees; they provide syntax for the category indicated by the top node. Initial trees fill substitution sites. The other type of trees, called auxiliary or *beta* trees, are spliced into initial trees through the TAG operation of *adjunction*. For instance, the tree shown in Figure 2.8 is an auxiliary tree which adds a prepositional phrase to a verb phrase. The foot node, indicated

<sup>&</sup>lt;sup>5</sup>The u: that is appended before the leaf node categories indicates that the information in these nodes can be given (or already known) as opposed to new. This is also unique to SPUD's use of LTAG.



Figure	Figure 2.8: Auxiliary tree $bVPpp(S, R, E, P, O)$			
NP(george)   George <sub>◊</sub>	$egin{array}{c} { m N(book)} \   \ { m book_{\diamond}} \end{array}$	$rac{N(\mathrm{floor})}{ }$ floor $\diamond$	$\overbrace{\mathrm{the}_\diamond}^{\mathrm{NP}(\mathrm{O})} \mathbb{N}(\mathrm{O}) \downarrow$	

Figure 2.9: Example lexical items

by the asterisk (\*), must be the same category as the top node and gives the location for the adjunction operation. In this case, adjunction applies to a node for a verb phrase, creating a subtree which consists of the verb phrase and the prepositional phrase. The same method is used to adjoin subordinate clauses to a main clause, as will be shown in Chapter 5.

As an example of how the tree operations work, consider the sentence "George moved the book to the floor." Lexical items needed for this example are shown in Figure 2.9, in addition to the lexical items past anchoring a simpleS tree (Figure 2.6), move anchoring a transitive VP tree (Figure 2.7), and to anchoring a bVPpp tree (Figure 2.8). The tree starts with a single substitution site for the type of tree desired; in this case, the initial tree is  $S(s,t,w1,george)\downarrow$  indicating a sentence tree with its semantic indices (to be explained in Chapter 5). The sentence tree anchored by past is substituted, giving the first tree shown in Figure 2.10. Next, the tree for the verb phrase anchored by move is substituted. At this point, the auxiliary tree anchored by to can be adjoined onto the tree. Notice that this adds an additional verb phrase (VP) node to the tree. These two operations result in the second tree in Figure 2.10. Finally, the noun phrases can be substituted in and



Figure 2.10: Construction of example sentence



Figure 2.11: Construction of example sentence (continued)

completed (see Figure 2.11). Morphological processing is needed to inflect the verb *move* to reflect the past tense feature dictated by the *past* lexical item (which should be erased). While this is a simple example, it shows how LTAG is used to represent lexical items and to combine them to form larger syntactic structures.

# 2.7 Natural Language Generation

"Text generation can be characterized as a process of transforming a message into a text. This process is successful if, and only if, the reader of the text is able to derive its intended message. The ultimate criterion of what it means for a text to be good is thus a cognitive rather than a strictly linguistic one: the easier it is for the reader to decode the intended message from the text, the better the text will be." [Scott and de Souza, 1990, p.47]

This section is meant as an overview of the structure of NLG systems, providing the framework for discussing how the implementation described in this dissertation compares with other NLG systems presented in Chapter 6. In addition, this overview brings together the issues discussed in this chapter and the previous one, by showing how the various topics fit into the generation process.

A generation system should take (or determine) communicative goals, goals to be achieved through the communication of information, and produce text which satisfy them. Generation systems must be given (or plan) the content to be conveyed and perform lexical choice and surface realization (refer back to Figure 2.5 for an overview of NLG system architecture). In order to carry out the transformation of goals into text, systems need a representation of the domain (e.g. concepts, objects, relations, etc.), a lexicon supplying words and their meanings, and a grammar providing ways of combining words into sentences (and possibly sentences into a discourse). Every system varies in their methods of content and text planning, lexical choice, and surface realization, and each uses different domain representations as well as lexicons and grammars. In this dissertation, I assume that by the time a system is generating a single instruction step, no further content or text structure planning is needed beyond choosing to use multi-clausal sentences or multiple sentences. So, leaving aside content and text structure planning, I focus my discussion of generation systems on their domain and lexical representations, their lexical choice methods and other aspects of their generation algorithms, and the quality of the texts produced.

**Domain and lexical representations** encode information about the domain, the lexicon, and the connections between the two. Similar to the representation of actions (Section 2.2), domain representation can be done in several formalisms, such as first-order logic, description logics, and feature structures. A key issue in domain representation is whether it is independent of linguistic considerations. A domain representation is *language-neutral* if it does not contain elements or structures that are required by any particular Natural Language. A related issue is the mapping of concepts in the domain to words in the lexicon. A one-to-one mapping between domain concepts and lexical items reduces the flexibility of generation since a concept will always be described in the same way. If the connection between concepts and words is many-to-many, there can be many different ways of relating the same concept in different contexts, precipitating the need for lexical choice. A critical issue in lexical representation is the inclusion of context in the representation of lexical items. That is, whether not only the meaning of a word or construction is represented, but also the context in which it has that meaning. This issue is important in terms of how lexical choice is done.

Lexical choice and realization algorithms are the tactical ("how to say it" as opposed to "what to say") components of a generation system — they perform *linguistic realization*, the transformation of semantics (meaning) into words and constructions. The variations in lexical choice algorithms were discussed in Section 2.4. While the lexical choice algorithm is a defining difference between generation systems, several other related differences exist. For instance, if a generation system uses a *lexicalized grammar*, one in which every piece of the grammar is associated with at least one word, then lexical choice performs the surface realization as well. Without a lexicalized grammar, a separate surface realization phase is needed to combine the chosen words into legal syntactic structures. The choice of a lexicalized or non-lexicalized grammar affects the lexical choice algorithm,

dictating whether lexical choice will choose words alone or words along with the constructions which they anchor. One final issue is whether backtracking, undoing a previous choice or decision, is used when legal sentences cannot be generated at first. Backtracking can occur within the lexical choice algorithm itself, usually when a lexicalized grammar is used, or during the surface realization phase, at which point the lexical choice phase must be redone. Finding a mapping from the semantics to a surface realization represents a search problem and differences in search algorithms are therefore applicable to lexical choice and generation algorithms.

Assuring the sensitivity, efficiency, and effectiveness of generated texts is essential for a successful generation system. Texts need to be *sensitive* to what the hearer knows. Different texts conveying the same information should be generated for hearers with different knowledge, tasks, etc. This could include making sure to use only words which the hearer knows (see [McKeown *et al.*, 1993]) or actions which the hearer is able to perform. Texts also need to be *efficient* by avoiding redundancy. In order to produce efficient texts, the generation system needs to be able to check which of the goals have been already achieved by the text at various points in the generation process. Among other benefits, this allows constructions to contribute to more than one goal (see [Stone and Webber, 1998]). Finally, texts need to be *effective*. They need to identify referents (objects, states/conditions, events, and actions) unambiguously and sufficiently to serve the communicative goals (in the case of instructions, enabling the correct performance of an action).

As will be discussed in Chapter 6, the issues raised here with respect to the generation process are addressed to varying degrees by previous NLG systems. None, however, fully address all of the issues, particularly those of expressing termination information and generating effective instructions. In the next chapter, the first step in generating effective instructions is discussed: namely, the study of naturally occurring instructions to determine how they convey termination information.

# Chapter 3

# **Expressing Action Termination**

Carrying out instructions relies on having all of the information about the actions in instructions, especially when to *stop* performing each action. Action *termination* information, therefore, needs to be available in some form in instructions. To see how instructions express termination information, I gathered examples of complex actions from several sources, including an F-16 aircraft maintenance manual and a Reader's Digest "do-ityourself" home maintenance manual. I expected to find that a significant number of the actions described in the instructions would have explicit expressions of action termination, since the intent of instructions is to have the reader carry them out. What I found confirmed this and, by examining how termination expressions are used in these instructions, I gained insight into how action termination is expressed for a variety of action and verb types.

While the most frequent source of termination information in the corpus of examples is the inherent culmination found in *accomplishment* and *achievement* verbs (see Section 2.1), a third of the instructions (around one thousand) have termination information coming from sources other than the verb. This means that enough interesting termination expressions are available to draw conclusions from with respect to common ways of expressing termination outside the verb. Coding each corpus instruction for action type and source of termination shows how frequently termination expressions appear and with which action types they tend to occur. The corpus analysis focuses on certain expressions in more depth, characterizing them in terms of how they provide termination information and what distinguishes them from other expressions. This sort of characterization is needed for automating the generation of such expressions correctly, among other tasks.

In the next section, I describe the corpora used for the corpus analysis and in Section 3.2 I show how the corpus is coded. (Appendix B contains a selection of the coded corpus.) Section 3.3 provides results and analysis of the coded corpus, including detailed analysis of selected termination expressions. The chapter ends with concluding remarks about how termination is expressed and how the information gained from the analysis is used in the rest of this dissertation.

# 3.1 About the Corpora

The corpora include the Reader's Digest New Complete Do-It-Yourself Manual [Reader's Digest, 1991], a version of the Organizational Maintenance Job Guide (Fuel System Distribution, USAF Series F-16C/D Aircraft) [USAF, 1988], which is a set of technical orders for the maintenance of F-16s, and a set of instructions for a virtual mitre saw assembly line [ITL SIMA, 1997]. This last corpus contains only numbered instructions, with no paragraph-length sections, and is meant as actions to be carried out by (virtual) workers on an assembly line. In addition, only the numbered instructions in the other two corpora are considered. Such step-by-step instructions are recognized as a sub-genre of instructions manuals by virtue of their distinguishing linguistic characteristics [Hartley and Paris, 1996]. Thus, focusing on the step-by-step portions of the corpora is well-motivated.

Since the corpora are meant to be used as guides to performing maintenance tasks as well as repairs and initial installations, they contain examples of concrete actions and the subset of step-by-step instructions provides imperative, not descriptive, sentences about the actions. The complexity of the sentences is restricted since each *instruction step*, i.e. set of sentences about a subtask, describes only one or two actions at a time. However, the variety of linguistic constructions (ways of expressing information) is large enough for the purpose of the corpus study.

The step-by-step subset of the corpora contains 3282 imperative main clause verb phrases (not necessarily sentences since conjoined main clauses are treated as two separate main clauses). Many of the instructions also contain subordinate clauses, which means that all of the verb phrases in the corpus outnumber just the main clause verb phrases. As described in the next section, each verb phrase, whether main or subordinate, is coded to indicate its type, its source of termination information, and its relationship to other verb phrases.

# 3.2 Methodology

The purpose of the corpus analysis is to identify sources (e.g. particular linguistic constructions) of termination information in maintenance instructions. The corpus analysis is not meant to discover exactly how termination sources, both linguistic and non-linguistic, provide termination information or how multiple termination sources interact. The coding of the corpus is kept simple, especially since it is done entirely by hand.

The simplest source of termination information is the verb itself (see Section 2.1). Verbs such as *remove* and *open* represent actions that have inherent culmination. Since simpler is better in Gricean terms [Grice, 1975], it is not unreasonable to expect that verbs will carry most of the burden of conveying termination information. However, when verbs for those actions which do not have inherent termination, such as *turn* and *slide*, are used, termination information must come from outside the verb. In these cases, termination information can from arguments of the verb (required or not) and from additional phrases and clauses. Having one verb phrase is simpler than having multiple verb phrases, so termination sources involving just one verb phrase are expected to be preferred, when possible, over multiple verb phrases. In general, information (whether about actions or not) is expected to be presented in the simplest and clearest way possible.

With these observations and expectations as a guide, the corpus is coded as follows. Each imperative verb phrase, whether main or subordinate, is given one of the following main codes:

IC (Inherent Culmination) — The verb has inherent culmination. Culmination is defined by the termination of an action or event accompanied by a characteristic change of state (see Section 2.1). Therefore, verbs with inherent culmination (such as remove and open) have termination information. The following examples illustrate IC verb phrases:<sup>1</sup>

- (10) a. Check fuse or circuit breaker box; reset tripped breaker or replace
   blown fuse. [Reader's Digest, 1991] IC,IC,IC
  - b. **Remove** safety wire and **disconnect** two hydraulic tubes from FFP hydraulic motor. [USAF, 1988] **IC**,**IC**
  - c. Get a trunnion and set it in the fixture. [ITL SIMA, 1997] IC,IC
- AC (Acquired Culmination) The verb phrase acquires culmination from some source other than the verb. The acquired culmination comes from the combination of the verb and its arguments and/or additional phrases and clauses, but the verb by itself does not have culmination. Since culmination is acquired from somewhere other than the verb, additional codes (described later) appear in the coding of the following examples:<sup>2</sup>
  - (11) a. Glue panels *together* with white or yellow glue. AC[adv]
    - b. Apply leak detection compound *around* fuel tank access panel. AC[pp]
    - c. **Press** the 2 buttons at the same time to **press** the parts together.

AC[pc-to:AC[adv]]

- AT (Acquired Termination) The verb phrase acquires only termination (i.e. it does not gain a characteristic change of state) from some additional phrase or clause; the verb does not have any termination associated with it. For example:
  - (12) a. **Pull** in one direction to refine cutting edge. AT[pc-to:IC]
    - b. **Bleed** *until* fluid stream is free of air. **AT**[until]
    - c. Push hard *when* putting the armature into the gear case. AT[when:IC]

<sup>&</sup>lt;sup>1</sup>From here on, examples from the corpus will follow the same pattern (unless otherwise noted), with **a** being from [Reader's Digest, 1991], **b** being from [USAF, 1988], and **c** being from [ITL SIMA, 1997]. The coding of each verb phrase in an example is given, in order, in the list of codes following the example.

 $<sup>^{2}</sup>$ When an additional clause has an action that is meant to be accomplished (as in a purpose clause), the coding of verb phrase that appears in the additional clause is given after the code for the additional clause and a colon.

In addition to these main codes, other codes indicate the additional phrases and clauses which appear with verb phrases. Multiple additional codes can appear with a main code. The codes for additional *phrases* within a verb phrase are:

- p (verb particle) The verb and a verb particle together have culmination or termination information. Verb particles can alter a verb's type so that the verb with the particle could be considered a separate verb, but verb phrases are coded based on the verb without its particle. Some examples include:
  - (13) a. Take up any water soaked carpeting and the padding underneath. IC[p]
    - b. **Turn** off air source. AC[p]
    - c. Screw in the third set screw. AC[p]
- arg (verb arguments) This code indicates that a verb argument other than a prepositional phrase (see the next code) provides culmination or termination information in conjunction with the verb. For instance:
  - (14) a. Cut other pieces 3 1/2 in. shorter than long pieces. AC[arg]
    b. Apply 60-70 psig air pressure to ground air service connector. AC[arg]
    c. Lower the arm. AC[arg]
- **pp** (**p**repositional **p**hrase) A prepositional phrase in a verb phrase gives culmination or termination to the verb phrase. Here are some examples:
  - (15) a. Fill a deep hole with gravel to within 4 in. of surface. IC[pp]
    - b. **Remove** coupling and **slide** sleeve *on* outlet tube. **IC**,**AC**[pp]
    - c. Slip an O-ring *onto* the locking pin. AC[pp]
- adv (adverb) An adverb or adverbial phrase in a verb phrase provides culmination or termination information. Examples include:
  - a. Insert a bit *fully* into the chuck. IC[adv]
    b. Screw boltheads [sic] *flush with* surface of protective frame. AC[adv,pp]
    c. Spray and wipe the unit *clean*. AC[oa], AC[adv]

Additional subordinate clauses are coded as to their type:<sup>3</sup>

- pc-x (purpose clause using lexical item x) A clause expressing a purpose-like relation between actions provides culmination or termination information to a main verb phrase. This clause need not be the linguistically-proper purpose clause of the form "to VP" (see Section 2.4). The x in the code represents the lexical item used in the purpose clause: to, as in the standard purpose clause; by, for the means clause (as it is linguistically named); and st, standing for such that or so that. Examples of purpose clause codes are:
  - (17) a. Mix mortar (p.165) so that it is a little stiffer than bricklaying mortar. AC[pc-st]
    - b. Depress bleed valve sufficiently to obtain stream of fluid flow.

AT[pc-to:IC]

c. Adjust the bevel pointer by tapping on it with a screwdriver until it points to 45 degrees.
 AC[pc-by:AT[until]]

Notice that for to and by clauses, the subordinate verb phrase is also coded.

Means clauses are included in the code for purpose clauses because they express a purpose-like relation, namely that doing the actions in the subordinate by clause achieves the action in the main clause. Thus they indirectly express a purpose and are included with purpose clauses.

- fa (free adjunct clause) Related to the purpose clause since they often convey purpose, free adjunct clauses can provide or modify culmination or termination information or modify the manner of the action by indicating a concurrent or purposive action for the action in the main clause. The following are some examples, all from [Reader's Digest, 1991]:
  - (18) a. Place blade assembly (bevel down) on the frog, engaging lateral adjusting lever. IC[fa:IC]

<sup>&</sup>lt;sup>3</sup>Other types of additional clauses are also coded for, such as *before*, *after*, *when*, *while*, *as*, and *for*, but they are not addressed here as they appear in less than one percent of the coded corpus.

- b. Then drill a shank hole, stopping at tape. AC[fa:IC]
- c. With firm pressure, draw round shank screwdriver along edge, forming flat, even burr.
   AT[pp,fa:IC]

Free adjuncts are complex constructions, which may explain why they are only found in the more complex portion of the corpus, namely [Reader's Digest, 1991].

- until (until clause) An until clause is a simple way to convey termination for the action in a verb phrase. In the case of instructions, the meaning of until is that an action is performed up to the time in which the state of the world expressed in the until clause comes into being. (For actions that inherently have culmination, adding an until clause changes the culmination.) For example:
  - (19) a. Drill through assembled (but unglued) joint until bit just touches tenon.
     AC[pp,until]
    - b. (C,D) Monitor fuel indicator until indicator reads 150-400 pounds in each reservoir.
       AT[until]
    - c. While holding the brush in, insert it into the slots of the field case until the brush is free to pop out and make contact with the commutator.

IC[while: AC[oa], until]

Finally, the code **oa** (other action) indicates that termination information for an action is provided through an inferable but non-lexicalized connection with another action in the instruction step. For example:<sup>4</sup>

(20) a. **Rotate** pipe slowly and **tap** chisel with hammer *until* pipe breaks off.

 $\mathbf{AT}[oa], \mathbf{AT}[until]$ 

b. Slide valve aft and remove. AC[oa],IC

c. Get a base assembly and lock it down on the table by pushing the button.

 $\mathbf{IC}, \mathbf{IC}[\text{pc-by:}\mathbf{AC}[\text{oa}]]$ 

<sup>&</sup>lt;sup>4</sup>To simplify the coding process, not to mention the notation, the action which provides the termination information is not indicated as part of the **oa** code. In the last example, the pushing action acquires its culmination from the accomplishment of the locking action but this is not noted explicitly in the coding.

Code	Description
AC	Verb phrase $\mathbf{A}$ cquires $\mathbf{C}$ ulmination
AT	Verb phrase $\mathbf{A}$ cquires $\mathbf{T}$ ermination
IC	Verb phrase has Inherent Culmination
adv	An $\mathbf{adv}$ erb adds or modifies culmination/termination
arg	Verb <b>arg</b> ument contributes to termination
fa	A free adjunct contributes to culmination/termination
oa	Relationship to other action provides termination
$\mathbf{pc-}x$	Termination from <b>p</b> urpose <b>c</b> lause using lexical item $x$
р	Verb $\mathbf{p}$ article indicates culmination/termination
pp	A <b>p</b> repositional <b>p</b> hrase supplies termination
until	An <b>until</b> clause provides termination

Table 3.1: Summary of codes used in corpus analysis

All of the codes described above are summarized (alphabetically for reference) in Table 3.1. The next section reveals the results of the coding.

# 3.3 Results and Analysis

The results of the coding bears out the hypothesis stated in the previous section: simple sources of termination information are more frequent. Overall, **IC** verb phrases represent two-thirds of the main clause verb phrases (see Table 3.2) and the remaining third of the corpus involves main clause verb phrases which acquire at least termination information (**AC** and **AT** verb phrases). The frequency of the additional codes (Table 3.3) shows that the simpler ways of adding termination information are more frequent: **pp** and **arg** do not involve an additional clause. Table 3.3 lists the frequency of the additional codes over all of the verb phrase types as well as the frequency with only the **AC** and **AT** verb phrases. The fact that purpose clauses are frequent stems from their multi-purpose nature, as discussed later in Section 3.3.3.

The distributions of codes for the different corpora sources show their different styles, with the main difference being the percentage of **IC** verb phrases (see Table 3.4). The F-16 corpus has the highest percentage (83%) of these, the simplest source of termination

Main VP code	Frequency	(%)
IC	2188	(66.7)
$\mathbf{AC}$	869	(26.4)
$\mathbf{AT}$	225	(6.9)

 Table 3.2: Overall Frequency of Main Clause Verb Phrase Codes

Add'l	Overall		In $AC/AT$	
code	Frequency	(%)	Frequency	(%)
pp	433	(13.2)	423	(38.7)
pc	308	(9.4)	202	(18.5)
arg	222	(6.8)	222	(20.3)
р	150	(4.6)	39	(3.6)
oa	91	(2.8)	91	(8.3)
adv	90	(2.7)	63	(5.8)
fa	79	(2.4)	39	(3.6)
until	55	(1.7)	47	(4.3)

Table 3.3: Frequency of Additional Codes

information. As a technical order manual whose instructions must be carried out as if they were direct military orders, the F-16 corpus reflects the philosophy that the easier it is to understand what needs to be done the better, since any mistakes could be costly. The Reader's Digest (RD) corpus has the widest range of constructions in its instructions, reflecting the variety of tasks that it encompasses. It has the highest percentage (45%) of combined **AC** and **AT** verb phrases, indicating the relative balance between the main verb phrase types. One similarity between the corpora is that they all have roughly the same percentage (between 12% and 15%) of verb phrases appearing with **pp** codes. However, within the **AC** verb phrases, the appearance of **pp** codes varies widely (only 34% in the Reader's Digest and 90% in the F-16 corpus). The distribution of codes gives a sense of the goals of the different corpora, i.e. the F-16 corpus is simple and straightforward so that following the instructions easy while the Reader's Digest uses powerful constructions to express quickly the most information possible.

Code	RD	F-16	SIMA
IC	919	1075	194
pc	98	2	6
$_{\mathrm{fa}}$	40	0	0
р	33	0	0
$\operatorname{adv}$	27	0	0
pp	6	4	0
until	6	1	1

Code	RD	F-16	SIMA
AC	587	181	101
$^{\rm pp}$	204	164	47
$\operatorname{arg}$	201	5	14
р	91	5	20
$\mathbf{pc}$	81	5	8
$\operatorname{adv}$	57	1	4
$_{\mathrm{fa}}$	28	0	0
oa	27	3	11
until	9	0	0

Code	RD	F-16	SIMA
AT	188	27	10
$\mathbf{pc}$	93	11	4
oa	44	3	2
until	25	12	1
$_{\mathrm{fa}}$	11	0	0
pp	8	0	0

Table 3.4: Code frequencies by corpora and main code

The fact that additional codes appear with **IC** verb phrases, especially in the Reader's Digest corpus, indicates the possibility of *modifying* the inherent culmination information that comes from verbs. In addition, multiple additional codes can appear with one main verb phrase, all interacting to provide the termination information for an action. The modifications and interactions are not considered in the analysis and Table 3.4 shows the total number of occurrences of codes, regardless of whether they are in combination with others. In the discussion that follows about use of constructions indicated by additional codes, co-occurrence with other constructions is not addressed.

#### 3.3.1 The use of verb arguments, particles, adverbs, and free adjuncts

Verb particles, adverbs, and free adjuncts are not relied on frequently to provide action termination, possibly because the interaction between the verb and these other components is complex. Verb arguments, appearing as a source of termination in only 7% of the instructions overall (but 20% of the **AC** verb phrases), are an especially complex source of termination information. How verbs and their objects can combine to give a culmination depends on the particular verb, the action it represents, and the type of its objects. The termination information from adverbs also relies on the particular verb, although certain adverbs can provide all of the termination information. Free adjuncts are very complex, relying on complicated relationships between the main action and the subordinate action

in the free adjunct clause. Finally, verb particles can change the type of the verb and, together with the verb, can act as a new verb, usually one with inherent culmination. The complexity of the interaction between verb and its arguments, adverbs, and free adjuncts, as well as the indivisible behavior of verbs and their particles, suggests that they are not termination sources to focus on when looking for semantically simple (and therefore more readily implementable) ways of expressing termination.

#### 3.3.2 The use of prepositional phrases

The greatest use of prepositional phrases comes in the **AC** verb phrases; their use in the other verb phrase types is infrequent. Prepositional phrases can provide culmination in terms of the endpoint of a path or a resulting configuration. They mostly appear with verbs describing motion which have no inherent culmination ([Badler et al, 1998; Bleam *et al.*, 1998] demonstrate this, especially for the F-16 corpus [USAF, 1988]). Both the endpoint of a path and a resulting configuration express a culmination: a path endpoint describes a new location for an object and a resulting configuration correlates with a more general change of state (also shown by [Dang *et al.*, 1998]). In fact, research in lexical semantics such as [Dang *et al.*, 1997; Palmer *et al.*, 1997] has shown that prepositional phrases are used to extend the meaning of many verbs in a regular way; that is, semantic information (e.g. an end configuration) correlates with syntactic behavior (e.g. prepositional phrases). These facts rule out the frequent use of prepositional phrases with verb phrases that already have a culmination or only acquire termination. Based on their most frequent use, then, prepositional phrases can be characterized as providing an action with a culmination involving an end configuration (geometric, spatial, or otherwise).

#### 3.3.3 The use of purpose clauses

Much work has been done in the area of characterizing purpose and means clauses (see Section 2.4). The corpus analysis presented here agrees with many of the observations made previously. Since purpose clauses are popular for a variety of uses, they are found in all three verb phrase types (see Table 3.5). Their use in **IC** verb phrases is to provide manner information or to modify the culmination provided inherently by refining or clarifying it. In

Code	%  w/pc
IC	4.8
$\mathbf{AC}$	10.8
$\mathbf{AT}$	48.0

Table 3.5: Co-occurrence of verb phrase types with purpose clauses

%	to	by	st
Overall	85.4	9.7	5.5
in main <b>IC</b> VP	76.4	23.6	0.0
in main $AC$ VP	78.7	5.3	15.9
in main <b>AT</b> VP	99.1	0.0	1.8

Table 3.6: Purpose clause distribution by lexical item and main verb phrase code

**AC** verb phrases, they are less frequent than other types of additional phrases and clauses; motion verbs that appear in **AC** verb phrases have simpler choices, such as prepositional phrases, for gaining culmination information. For **AT** verb phrases, purpose clauses are the most frequent way of acquiring termination information. By giving the purpose for doing an action that does not inherently have termination, the action's termination is understood to coincide with the fulfillment of the purpose. In this way, both the high-level action (i.e. the purpose) and how to accomplish it are given in the same sentence, making an efficient presentation of information.

The *kind* of purpose clause also varies with the verb phrase type it co-occurs with. The most frequent kind of purpose clause is the standard *to* purpose clause, however as shown in Table 3.6, the *by* or *means* clause (see Sections 3.2 and 3.3.3) is also well-used. Below I present examples from the corpus and discuss possible formalizations of the use of purpose and means clauses.

Instructions which use means clauses present the task to be accomplished (i.e. the purpose) first and then the means (i.e. the actions) by which it can be accomplished. This can be seen in the examples, all from [Reader's Digest, 1991], shown in Figure  $3.1.^5$ 

<sup>&</sup>lt;sup>5</sup>Remember that the code for a subordinate verb phrase appears after the additional code for the type

- (21) a. **Empty** toilet bowls and tanks by siphoning or bailing and sponging. IC[pc-by:AC[oa]]
  - b. Assemble horse by sliding legs *into* channels formed by saddle's stop blocks. IC[pc-by:AC[pp]]
  - c. Level bricks by tapping lightly with a rubber mallet. IC[pc-by:AT[oa]]
  - d. To check the wall's batter (slope), make a batter gauge by nailing three 1 x 2's to form a 90 degree angle. IC[pc-to:IC,pc-by:AC[pc-to:IC]]
  - e. **Unscrew** float by **turning** it counterclockwise on float arm. IC[pc-by:AT[oa]]

Figure 3.1: Examples of by purpose (means) clauses

The task is (nearly) always one with an inherent culmination and the means are (nearly) always actions which acquire culmination or termination, usually from the task for which they are done. Using a means clause is a straight-forward way of presenting a high-level action and then its sub-actions.

Standard purpose clauses, those introduced with to, can appear fronted or non-fronted (see the discussion of [Thompson, 1985] in Section 2.4). In the case of the non-fronted purpose clauses, they can provide the manner or termination (or both) of the action they modify in addition to a purpose. Figure 3.2 gives some examples, again all from [Reader's Digest, 1991], of non-fronted purpose clauses. In these cases, the actions expressed in the purpose clause are usually more abstract than those expressed in the main clause of instructions using means clauses. The low-level actions are essential to accomplishing the task since they provide concrete details and thus are placed before the purpose clause.

In contrast to the non-fronted purpose clauses, fronted purpose clauses place highlevel actions before the lower-level actions which accomplish them (see Figure 3.3). The purpose action in the fronted purpose clause is even more abstract than in the non-fronted purpose clause, describing a generic action or overarching goal. In addition, the lower-level

of the higher clause. Thus, pc-to:IC means that to purpose clause contains an IC verb phrase.

(22)	a.	Glue and nail one long and one short piece to form	<b>n</b> a unit.
			$\mathbf{AC}[\mathrm{oa}], \mathbf{AC}[\mathrm{pc\text{-to:}}\mathbf{IC}]$
		<b>Align</b> all edges; then <b>apply</b> clamps and weights $to$ glue dries.	<b>maintain</b> position <i>until</i> IC,AT[pc-to:AT[until]]
	b.	<b>Pull</b> in one direction to <b>refine</b> cutting edge.	$\mathbf{AT}[ ext{pc-to:}\mathbf{IC}]$
		Set blade to cut through work and barely into scrap	$\mathbf{IC}[\text{pc-to:}\mathbf{AC}[\text{pp}]]$

Figure 3.2: Examples of (non-fronted) to purpose clauses

actions are either two (or more) in number or complicated. The difference between fronted purpose clauses and means clauses, which also put the high-level action first, is that the high-level action in fronted purpose clauses is more abstract and serves as an introduction to a complicated sequence of actions to achieve the purpose.

#### 3.3.4 The use of *until* clauses

The use of *until* clauses for providing termination information is not as common as expected. They are a simple way of conveying termination, however that is all they typically do. When another phrasing is possible, such as a prepositional phrase or a purpose clause, an *until* clause is not used. For instance, the instruction *Turn the dial until it is at the ON position* would not be used instead of the instruction *Turn the dial to the ON position*, even though the former conveys the proper termination. Unlike purpose clauses or prepositional phrases, an *until* clause does not necessarily state explicit connections between the action and its termination information (e.g. a purpose relation or an end configuration of a manipulated object). An *until* clause is capable of stating a completely unrelated termination condition for an action, as in "Do your homework until your mother gets home." Although no such examples appear in the corpus, the reader must still do extra reasoning to understand how the termination condition in the *until* clause relates to the action in the main clause. This burden on the reader could explain why such a simple termination source is infrequent, used in only 1.7% of the corpus despite appearing with nearly half of

- (23) a. To fix small blisters in linoleum or soft vinyl flooring, puncture them with a nail, and pump epoxy through nail hole, using a glue gun (p.90) with a syringe, or hypodermic, nozzle. IC[pc-to:IC], AC[pp,pc-to:IC]
  - b. To help you keep the drill straight, position or clamp a try square or combination square near the drill and keep the drill parallel to the square. IC[pc-to:AC[oa]],AC[oa],AC[oa]
  - c. To make a hole of the depth you want, use a commercial drill stop or gauge, or wrap a piece of masking tape at the appropriate height on the bit.
     AC[pc-to:IC],IC[pc-to:IC]
  - d. To use circular saw for cutting grooves and dadoes (p.102), mark the width and depth of the cut. IC[pc-to:AC[arg]]
  - e. To bend thick plywood, make saw kerfs just below the face ply at 3/16 to 1/4 in. intervals. IC[pc-to:AC[oa]]
  - f. To mark tails, first scribe shoulder line 1/32 in. wider than pin piece's thickness. AC[pc-to:IC,arg]
  - g. To cut joint fingers, place stock against guide block and carefully push entire assembly over turning dado head. IC[pc-to:AC[arg]],AC[pp]

Figure 3.3: Examples of fronted to purpose clauses

the F-16 AT verb phrases.

## 3.4 Conclusion

The focus of this corpus analysis has been on the more frequent and semantically simpler sources of termination and how they are used. These include prepositional phrases for expressing the endpoint of a motion's path, purpose clauses for tying low-level actions with high-level ones, and *until* clauses for stating termination conditions of actions. These have been analyzed in detail since they are semantically simpler than other termination sources, such as verb arguments and adverbs. However, except for *until* clauses, they are common in maintenance instructions and thus provide a good basis for generating termination expressions.

The corpus analysis has been used both in the development of an action representation which supports termination information as well as the formulation of rules for determining whether a particular action has termination information. These are described in the next chapter. In Chapter 5, the encoding of how additional phrases and clauses contribute particular action information is shown and used by the Natural Language generator to generate complex instructions involving expressions of action termination.

# Chapter 4

# Representing Objects, Actions, and Agent Expertise

The Natural Language generator SPUD (Sentence Planning Using Descriptions) [Stone, 1998] reasons about Natural Language Generation (NLG) tasks using a logic theoremprover [Stone, 1997], presenting the opportunity to develop domain knowledge which is independently-motivated. Implementing a domain model in SPUD is much like writing a logic program since it uses modal first-order logic, a combination of first-order logic and modal logic (see Section 2.3). Therefore, SPUD can be seen as a programming environment for implementing domains as well as generating text.

The implemented domain model encompasses several sources of information, including information about objects, actions, and agents. Object knowledge includes information about objects' properties and connections to other objects in the domain. Action knowledge represents general and specific information about actions in the domain, including how actions interact with other aspects of the domain. Agent knowledge reflects what agents and types of agents are assumed to know about the domain and its actions. In particular, it models different agent expertise levels. After the description of the example domain considered in this dissertation, the rest of the chapter discusses the representation of the domain knowledge in SPUD.



Figure 4.1: A control panel for a pump

# 4.1 The Example Domain

Inspired by the tasks found in the F-16 aircraft maintenance instruction manual (see the previous chapter), the example domain involves a control panel for a pump (see Figure 4.1). The control panel has a lever which resets the pump, a button which controls the on/off state of the pump, a light which indicates the pump's state, a dial which controls the pump's pressure, and a gauge that indicates the pump's pressure. The panel also has a cover which is held in place by a screw.

This domain provides varied tasks which allows a wide range of action and agent information. Tasks in the domain form a collection actions and sub-actions, of which Figure 4.2 reflects a small portion. The actions can be high-level, such as normalizing the pump's pressure, and have sub-actions which are lower-level, such as turning the dial. Highlevel actions typically have inherent culmination (see Section 2.1) whereas low-level actions usually do not. Therefore, a wide variety of termination information, action information which indicates when to stop performing an action, is needed for many actions in the domain. In addition, less experienced agents are less likely to know how to perform highlevel actions without explicit instructions, but they are likely to know how to perform low-level actions on their own. Thus, the knowledge that agents can be assumed to have about actions in the domain matches the wide array of action information possible in the



Figure 4.2: Portion of action/sub-action tree in the control panel domain

domain. Before action and agent knowledge can be represented, however, information about objects and their roles in the domain must be specified.

# 4.2 Object Information

Object information includes information about specific objects in the domain as well as general information about classes of objects. Each of the specific objects, physical or abstract, must be described in terms of its type, its place in a part/whole tree, and its properties, including its connections to other objects in the domain. Physical objects are those that can be directly manipulated, such as screws, dials, and levers. Abstract objects can be identified in the domain, but cannot be directly manipulated. (Examples of abstract objects will be given later.)

For example, consider the screw, called screw1, which holds the cover on the panel. The first two pieces of knowledge in Figure 4.3 indicate screw1's type. It is a physical object and its type is a standard screw.<sup>1</sup> Since the representation is a *modal* FOL (mFOL), the modal status of each piece of information needs to be given. In this case the modal status of each piece is that of *common knowledge*, indicated by the C modal operator, meaning that the information is assumed to be known by all agents in the domain. (More will be said about modal operators in Section 4.4.1.) The third item indicates the screw's

<sup>&</sup>lt;sup>1</sup>The representation of objects presented here is only one possible way among many. The specific representation chosen in this dissertation is not essential to the results.

```
C physicalObj(screw1).
C type(screw1, standardScrew).
C partOf(screw1, cover1).
C property(screw1, state, oneOf(tight,loose)).
C property(screw1, defaultLocation, in(hole1)).
C property(screw1, loosenDir, ccw).
C property(screw1, tightenDir, cw).
C property(screw1, turnable, true).
C property(screw1, unique, true).
```

Figure 4.3: Knowledge about the screw



Figure 4.4: Part/whole object tree in the control panel domain

relationship to other objects in the domain, namely as part of the cover which in turn is part of the panel. Figure 4.4 shows a portion of the part/whole tree which is defined by the partOf and hasPart predicates.

The remaining items of information in Figure 4.3 give static and dynamic properties of the screw. The state of an object is a dynamic property and, in this case, two states are possible: tight or loose. Another dynamic property of a movable object is its current location, but since this can range over all possible locations in the domain, it is left as an implied property. Unlike an object's current location, its default location, or where it would be assumed to be given no other information, is a fixed property; the screw's default location is in the hole (hole1). (Locations are simply represented by a relation name, such as in, on, or at, combined with the object identifier, e.g. in(hole1)). Two more fixed properties, namely those for the loosening and tightening directions, are also specified (counterclockwise and clockwise, respectively). The last two pieces of information about the screw are also static properties: turnable, indicating that it can be turned, and unique, indicating that it is the only such object (i.e. with the exact same properties) in the domain.

In addition to this statement-like knowledge, knowledge can be formulated as rules in mFOL. For instance, a rule can state that all objects with the type **standardScrew** also are of type **screw**:<sup>2</sup>

```
*O C(type(O, standardScrew) -> type(O, screw)).
```

This rule is used to reason that standard screws are screws.<sup>3</sup> For instance, using this rule, the fact type(screw1,screw) can be proven from the above knowledge about screw1. This allows other knowledge rules to state general properties of all screws without having to have a separate rule for each subtype.

The object information shown in Figure 4.3 is typical of most physical objects in the domain. However, the example does not demonstrate how connections between objects are represented. One connection is whether an object controls or is controlled by another object. For instance, the dial is one source of control of the pump's pressure. Thus, in the representation of the dial, it would have the following property:

```
C property(dial1, controls, pressure1).
```

indicating that it exerts control over the abstract object, **pressure1**. Likewise, in the representation of the pump's pressure, the following appears:

#### C property(pressure1, controlSource, dial1).

This states that the pump's pressure is controlled by the dial. The connections between objects can be represented and reasoned about by the theorem-prover using these types of properties. While these types of properties are not used in the current work, they would be needed for more sophisticated reasoning about the domain and thus are included here for further work.

<sup>&</sup>lt;sup>2</sup>\*0 means  $\forall O$  and  $\rightarrow$  is logical implication  $(\rightarrow)$ .

 $<sup>^{3}</sup>$ Again, this is only one possible way to reason about object types. Such details are not critical to, nor a focus of, this dissertation.

```
C abstractObj(restPos1).
C type(resetPos1, position).
C partOf(resetPos1, lever1).
C property(resetPos1, label, reset).
C property(resetPos1, location, above(defaultPos1)).
C property(resetPos1, movable, false).
C property(resetPos1, unique, true).
```

Figure 4.5: Object information for the lever's "RESET" position

Abstract objects include objects that are not usually thought of as objects. As noted above, the pump's pressure is represented as an abstract object since it is not directly manipulable. Other abstract objects include the "RESET" lever position and the "normal" range of the gauge. These are non-manipulable but identifiable (i.e. capable of being referred to) aspects of the domain. Figure 4.5 shows the representation of the "RESET" lever position. A property which appears in its representation which has not been mentioned previously is the label property, which indicates the identifying label of the object. This property is also used for physical objects with labels, such as the "ON" button.

The full representation of all of the objects in the domain and the rules to reason about them are given in Appendix A.4.

### 4.3 Action Information

Through the study of maintenance activities with a mixed group of human modeling and simulation (HMS) researchers and Natural Language researchers, action information needed for both purposes emerged. Work by others in the area of verb semantics [Badler et al, 1998; Dang *et al.*, 1997; Dang *et al.*, 1998] as well as the informational requirements of simulating agents carrying out maintenance activities [Badler et al, 1997], along with the analysis of maintenance instructions, contributed to the development of the action representation described below. While this dissertation work does not take advantage of all the capabilities of the action representation, I present its full extent (so far) in order to hint at the range of actions and events it can represent. The rules needed for reasoning about action information in terms of termination and concreteness given this representation are discussed in 4.3.2 and 4.3.3, respectively.

#### 4.3.1 Parameterized Action Representation (PAR)

For actions that are to be carried out or otherwise interpreted, their specification must include all of the necessary components. A representation called **PAR** (Parameterized Action Representation) has been developed with this in mind. It is meant to be a common representation for animating virtual agents as well as generating Natural Language instructions [Badler *et al.*, 1998; Badler *et al.*, 1999]. The structure of PAR is shown in Figure 4.6 and each component is explained below.<sup>4</sup> Those components appearing with an asterisk (\*) are not further addressed in this dissertation.

- \*Applicability condition is a boolean expression of conditions (conditions conjoined with logical *ands* and *ors*) which must be true in order for the action to be appropriate to perform. These conditions generally have to do with certain properties of the objects, abilities of the agent, and other unchangeable or uncontrollable aspects of the environment. Unlike a precondition (see below), it would be impossible or impractical to try to satisfy the applicability condition as a subgoal before performing the action.
- **During** is the time interval in which the action takes place.
- **Result** is the time interval after the action is performed.
- **Participants** are those entities participating in the action.
  - **Agent** is the animate entity who performs the action. The representation of the agent can include its physical attributes and its capabilities.
  - **Objects** is the list of entities/objects involved in the action. The representation of objects can include physical properties such as geometry and current state as well as actions defined for the objects. It is possible that the list could associate a role, such as **instrument**, along with an entity.

<sup>&</sup>lt;sup>4</sup>Although PAR appears here as a feature-structure representation, translation into mFOL is straightforward as described later.

PAR			
applicability condit	ion: CONDITION bool-exp		
during:	TIME-INT		
result:	TIME-INT		
participants:	agent:AGENTobjects:OBJECTlist		
core semantics:			
precondition: postcondition:	CONDITION <b>bool-exp</b> CONDITION <b>bool-exp</b>		
motion: ob, cai tra rot	ject: OBJECT used: BOOLEAN unslational: BOOLEAN tational: BOOLEAN		
force: ob po	ject: OBJECT int of contact: OBJECT LOCATION		
path: end: distar	tion: DIRECTION LOCATION LOCATION hee: LENGTH		
purpose: achie enabl	ve: CONDITION bool-exp ate: PAR e: PAR		
termination: CC	NDITION bool-exp		
duration: LENGTH			
manner: MANNER			
subactions: PAR constraint-graph			
previous action:	PAR		
next action: PAR			
concurrent action: PAR			
parent action:	PAR		

Figure 4.6: The structure of PAR

Core semantics represents the primary components of the action.

- \***Precondition** is a boolean expression of conditions that must be satisfied before attempting the action, in order for the action to be successful. Although disjunctions are possible, it is generally just a condition or conjunction of conditions. (The use of preconditions is the traditional method of subgoaling that is found in planning.)
- **Postcondition** is a boolean expression of conditions which holds after the action is done (i.e. in the **result** time interval). It generally predicates changes of state in object properties and/or relations between objects.
- Motion represents any motion component of the action. It is a substructure which indicates the object undergoing the motion and whether the motion is translational or rotational. (For the motion of objects by an agent, the **caused** component of the motion substructure will always be true. Since this dissertation addresses actions done by agents, the **caused** component is assumed to be true in all actions without being explicitly included.)
- **Force** represents any explicit force component of the action. It is a substructure containing the object to which the force is applied and the point of contact. (If the point of contact is omitted, a default point of contact can be assumed to come from the properties of the object.)

Path represents path information for the action.

- **Direction** gives the direction of any motion or force. Directions can be absolute or relative to an object or agent.
- \*Start indicates the starting location of the motion. Locations are generally represented by a relation (e.g. *on*, *at*) with an object (see the previous section).

 ${\bf End}\,$  indicates the end location of the motion.

\*Distance indicates the length along the path. A length consists of units (e.g. miles, degrees) and a quantity (e.g. 90).
- **Purpose** indicates the purpose of the action. The purpose can include a boolean expression of conditions to achieve (make true), an action to generate, and/or an action to enable.<sup>5</sup>
- \*Manner indicates any constraints, not otherwise represented, on the manner in which the action is to be done.
- \***Termination** indicates any termination conditions which would not be otherwise covered (e.g. by a post-condition, path endpoint, or purpose). This is needed for actions in which there is no relation between the action and the conditions except that the action is terminated when the conditions become true.
- \*Duration indicates any explicit duration for the action. It is similar to the distance component of **path** in that has units and a quantity. Although the units used for duration are usually those for time (e.g. *seconds, minutes*) and iteration, durations involving spatial units are also possible (e.g. "Drive for ten miles").
- **Subactions** represents the breakdown of the action into its sub-steps. It is a collection of actions connected in a graph structure which indicates the temporal relationships (if any) between the actions (e.g. whether two actions are to be done sequentially, in parallel, *etc.*). (For this dissertation, sub-actions are always sequential.)
- **Previous action** is an action done immediately before the action.
- **Next action** is an action which is done immediately after the action.
- \*Concurrent action is an action which is done in parallel with the action (as indicated by the parent action's sub-actions graph).
- **Parent action** is the action of which the particular action is a sub-step.

Translation of the PAR feature-structure (FS) representation into mFOL for SPUD could be automated. The FS representation is a simple, commonly used representation and, as such, is a good representation for knowledge that is used across applications.

<sup>&</sup>lt;sup>5</sup>For this implementation, the purpose of maintaining some condition can be loosely treated as an action to generate. For further work, PAR should be expanded to include maintenance purposes.

```
id = ua1
during = t_{1_1}
result = t_1_2
                 agent = u
participants =
                 objects = (screw1, instrument(screwdriver))
                                object = screw1
                    motion
                                type = rotational
core semantics
                   postcondition = configuration(screw1, state(loose))
path =
        direction = ccw
previous action = nil
next action = ua2
parent = oa1
```

```
∜
```

```
S during(ua1, t_1_1).
C result(ua1, t_1_1_2).
S agent(ua1, u).
S instrument(ua1, screwdriver).
S motion(ua1, screw1, rotational).
S pathDir(ua1, ccw).
S postcondition(ua1, configuration(screw1, state(loose))).
S prevAction(ua1, nil).
S nextAction(ua1, ua2).
S parent(ua1, oa1).
```

Figure 4.7: PAR example and its translation into mFOL

Creating mFOL statements from PAR is straightforward in general: FS attributes become predicates, action identifiers become the first argument to the predicates, and attribute values become additional arguments to the predicates.<sup>6</sup> For example, Figure 4.7 shows a PAR example for an action and its translation into mFOL. Since agents are not allknowing, actions need to be conveyed to them by the system; therefore, in the translation to mFOL, all but one piece of action information (namely, the fact that each action has a time interval following it) is listed as system knowledge using the modal operator **S** (described in the next section).

<sup>&</sup>lt;sup>6</sup>For reasons of efficiency and clarity of the implementation, the translation process is not purely syntactic although there is nothing fundamental to keep it from being so. For instance, adhering to the layered structure of PAR is not strictly necessary. A pseudo-code algorithm for the translation process used in this implementation is given in Appendix C.

```
*P *R C(starts(R,P) -> subinterval(R,P)).
*P *R1 *R2 C(meets(R2,R1), subinterval(R1,P) -> subinterval(R2,P)).
```

Figure 4.8: Rules for time subintervals

Since time intervals are associated with actions (i.e. the during and result slots in PAR), knowledge of how time intervals interact needs to be included in the domain knowledge. For the purposes of this dissertation, this knowledge includes three relationships between the time intervals (see Section 2.1). For the example action given in Figure 4.7, the following statements are also included in the action information:

S starts(t\_1\_1\_1, t\_1\_1).
S meets(t\_1\_1\_2, t\_1\_1\_1).
S finishes(t\_1\_1\_2, t\_1\_1\_1).

to represent the time interval structure in which an interval  $(t_1_1)$  has two subintervals which are contiguous.<sup>7</sup> Knowledge about time intervals also includes the transference of properties from intervals to subintervals where subintervals are represented by the rules in Figure 4.8.

#### 4.3.2 Reasoning about action termination

PAR represents action information needed for carrying out actions, including termination information. Some reasoning is required, however, to determine all of the necessary action information. For instance, if an action involving motion includes path endpoint information (e.g. "Turn the dial to the ON position"), then it has termination information, whereas a similar action without the endpoint information (e.g. "Turn the dial") does not. This reasoning is formalized by the first rule shown in Figure 4.9; it states that for all types of motion actions involving any object and a path endpoint<sup>8</sup>, termination information is

<sup>&</sup>lt;sup>7</sup>Note that the subinterval is listed first in the starts and finishes relations and the later interval is listed first in the meets relation.

<sup>&</sup>lt;sup>8</sup>pathEnd(E,P) is equivalent to the substructure [ path = [ end = P ] ] in the PAR feature structure for E. (See Appendix C.)

- 1. \*E \*O \*T \*P C(motion(E,O,T), pathEnd(E,P) -> termination(E)).
- 2. \*E \*P C(postcondition(E,P) -> termination(E)).
- 3. \*A \*P C(purpose(A,generate(P)) -> termination(A)).

Figure 4.9: Rules for reasoning about termination information

available. Other components of action information that provide termination include the **postcondition** and the **purpose** of an action and rules for these are also shown in Figure 4.9. The modal FOL which SPUD uses allows such reasoning to be represented, which is especially useful when determining what information an agent can be assumed to have about actions.

#### 4.3.3 Reasoning about concreteness of actions

In this dissertation, I take a concrete action to be one which includes the necessary information for performing the intended action. An action instance containing all the necessary information for its performance can be taken by a Natural Language generator to produce an effective instruction. If an action instance is not concrete, then its description will not be effective. For example, if it is intended for an agent to turn something clockwise, then direction information needs to be included in the action instance. This can be seen as an extension of reasoning about the termination information provided by an action. Similar rules to those given in the previous section are used to define what makes an action concrete. For the example given above, a rule can be stated that an action that involves motion needs path information in order to be concrete. This, along with other concreteness rules, are shown in Figure 4.10.<sup>9</sup>

The first rule in Figure 4.10 states that an action is concrete if it involves motion and has path information (where path information can be a direction (1a) or an endpoint (1b)). The second rule states that an action involving force is concrete if it includes information about its magnitude and direction. The next two rules involve purpose relations between

<sup>&</sup>lt;sup>9</sup>SPUD notation: ? means  $\exists$ , \* means  $\forall$ , and , (comma) means  $\land$  (and).

```
1. *A *O *T C(motion(A,O,T), path(A) -> concrete(A)).
   (a) *A *D C(pathDir(A,D) -> path(A)).
   (b) *A *D C(pathEnd(A,D) -> path(A)).
2. *A *O *W *D C(force(A,O), magnitude(A,W), pathDir(A,D) ->
   concrete(A)).
3. *A *O *T *P C(motion(A,O,T), purpose(A,P) -> concrete(A)).
4. *P C(purpose(P), (?A purpose(A,P)) -> concrete(P)).
5. *A C((?S subactions(A, S), concreteAll(S)) -> concrete(A)).
   (a) C concreteAll(nil).
   (b) *A C((?N nextAction(A, N), concrete(A), concreteAll(N)) ->
      concreteAll(A)).
```

Figure 4.10: Rules for concreteness

actions. If a motion action includes information about the purpose for which the action is done, then the action is considered to be concrete. Likewise, if an action is viewed as a purpose in general (i.e. purpose(P), explained in Section 5.2.1) and serves as the purpose for another action, then the action is concrete. Finally, if an action has sub-actions and they are all concrete, then the action itself is considered concrete. Although these rules are oversimplifications, they suffice and could be modified as necessary.

#### 4.4 Agent Expertise Information

Agent expertise information includes facts that agents are assumed to know as well as rules that they are assumed to use to reason generally and specifically about objects, actions, and behaviors in the domain. By using modal operators to represent different agent types, agent knowledge reflects levels of experience with the domain. In Section 4.4.1, modal operators representing three levels of agents for this domain are presented. While most of the general knowledge about objects and actions (described in the previous sections) is shared by all agent types, knowledge about specific objects and actions in the domain is assumed to be known by only certain agent types. Section 4.4.4 presents examples of how general and specific action knowledge, discussed briefly in 4.4.2 and 4.4.3, can be used to reason how much information about a specific action must be given to an agent of a particular type in order for that agent to understand how to perform the action.

#### 4.4.1 Agent Types

Agent types are represented by different modal operators. Knowledge predicated with each modal operator indicates the assumed knowledge of the corresponding agent type. The modal operators are defined as nesting, i.e. each operator encompasses the knowledge of the operator below it. The modal operators are as follows:

- **S** represents the system; knowledge predicated with this modal operator (for example, details of specific actions to be carried out) is private to the system.
- U3 represents the *advanced* agent type; this is the most knowledgeable (non-system) agent type.
- **U2** represents the *beginner* agent type; beginners have limited experience with the domain.
- U1 represents the *novice* agent type; this agent type is assumed to have no real experience with the specific domain, but may have knowledge gained from similar domains.
- C represents common knowledge; this modal operator is used for knowledge assumed to be known by all possible agents, not just those represented by U1, U2, and U3.

Viewing the modal operators as sets of information, the nested behavior of the operators could be seen as the following set relationships:  $S \supset U3 \supset U2 \supset U1 \supset C$ . This is shown graphically in Figure 4.11. For instance, a beginner agent would be assumed to have the knowledge predicated with U2 as well as the knowledge of novice agents (U1).

Using other modal operators, knowledge that is private to an agent type (i.e. not inherited by the agent types above it) or to a particular agent (determined by prior experience with the agent or through some other source) can also be represented if necessary. For general reasoning about actions, the nested operators are sufficient; however, as shown in

S	(system knowledge)
บ3	(advanced knowledge)
ับ2	(beginner knowledge)
<b>U1</b>	(novice knowledge)
C	(common knowledge)

Figure 4.11: Nested modal operators for agent types

the next chapter, non-nested operators are used in the generation of instructions for the different agent types.

#### 4.4.2 General knowledge

General knowledge about the domain encompasses general knowledge about objects, actions, and behaviors. As shown in Section 4.2, general knowledge about objects includes rules for reasoning about their types and properties. For example, the rule that states that a standard screw is a type of screw is included in general knowledge. Rules are also used for reasoning about properties of objects. The rule

```
*O C(property(O, turnable, true) -> property(O, movable, true)).
```

is an example of reasoning from a specific property (turnable) to a more general one (movable). Some of these rules could be restricted to certain agent types, but this implementation does not do so.

General knowledge about actions includes reasoning rules for the general properties of actions, such as the rules about termination and concreteness given in Sections 4.3.2 and 4.3.3. In addition, other general rules about actions are included; for instance, if an action has a post-condition involving the state of an object, then the post-condition will hold in the time interval following the action:<sup>10</sup>

<sup>&</sup>lt;sup>10</sup>The binary configuration notation is used to indicate generic states of objects, not tied to any particular time interval. The ternary configuration relation, on the other hand, specifies the particular time interval for which the the predicated state of the object holds.

# 

This knowledge is shared by all agents, but general action knowledge could be restricted to certain agent types, such as in the following rule:

```
*S *A U2(type(S,screw),
    postcondition(A,configuration(S,state(loose))) ->
    motion(A,S,rotational),
    (?D property(S,loosenDir,D), pathDir(A,D))).
```

This rule states that if a screw is to be loose as the post-condition of an action, then beginner agents (or higher) would know, just from that information, that the action involves rotational motion in the loosening direction of the screw.

Another type of knowledge about the domain is that of the interaction of time subintervals and the predicate **present** over time intervals. In this implementation, the rule

```
*T *S C(present(T), subinterval(T,S) -> present(S)).
```

is used along with the rules defining subintervals (see Figure 4.8) to allow the reasoning that subintervals of the present time interval can also be considered as the present. While other general knowledge about behaviors in the domain is not used in this implementation, rules such as those for the inertial behavior (in the frame problem sense) of objects and their properties could be included and used for reasoning about the domain.

#### 4.4.3 Specific knowledge

Specific knowledge about objects, actions, and behaviors reflects experience with the particular domain, as opposed to experience with similar domains (reflected in the general knowledge discussed above). Specific knowledge about objects and behaviors includes the types and properties of specific objects as well as any quirks that particular objects have (for example, the lever could be more difficult than usual to lift). In this implementation, I have assumed that all of the information about the specific objects in the domain is common knowledge. Such an assumption is not generally well-founded since novice agents cannot

```
*A (postcondition(A, configuration(panel1, state(open))) ->
    U3 ?S(subactions(A,S), nextAction(S,nil),
        postcondition(S, configuration(screw1,location(awayFrom(hole1)))))).
*A (postcondition(A,configuration(screw1,location(awayFrom(hole1))))
        -> U2 ?S?N (subactions(A,S),
            postcondition(S,configuration(screw1,state(loose))),
            nextAction(S,N), motion(N,screw1,state(loose))),
            nextAction(S,N), motion(N,screw1,translational),
            pathEnd(N,awayFrom(hole1)), nextAction(N,nil))).
*A *O (type(0,screw), motion(A,0,rotational) ->
        U1 instrument(A, screwdriver)).
```

Figure 4.12: Agent knowledge about opening the panel

be assumed to know the specifics of novel objects. Making certain object information part of a particular agent type's knowledge would solve this problem if such a distinction were useful. As for specific behaviors, none have been needed in this implementation to date, but rules expressing such behaviors could be easily added.

Specific knowledge about actions includes what actions constitute achieving or doing other actions in the domain. For instance, advanced agents might know that opening the panel involves removing the screw. Beginner agents might know what removing the screw involves but not that removing the screw is a step in opening the panel. Since the modal operators are nested, the predicated knowledge of the advanced agents (U3) does not need to include the knowledge of the beginning agents (U2), as shown in the representation of knowledge about opening the panel given in Figure 4.12.

Rules for other specific actions are expressed similarly and are discussed in more detail in the next chapter. The following discussion, however, presents a brief example of how the specific and general knowledge discussed so far is used to reason about an agent's knowledge about a specific action when presented with only certain information.

#### 4.4.4 Reasoning about agent knowledge

Knowledge that an agent has about the domain, as shown above, reflects the assumed experience level of the agent. This knowledge can be used in conjunction with rules about what an agent needs to know in order to perform an action (discussed previously in Sections 4.3.2 and 4.3.3). The aim of encoding this knowledge is to be able to query whether an agent knows how to perform a particular action as intended given certain information. For the sake of clarity, I will assume that the following rule:

#### \*A C(concrete(A), termination(A) -> howToDo(A)).

is in the domain knowledge and can be used to reason about whether an agent has enough information to know how to do an action.<sup>11</sup>

Consider the action of opening the control panel. The agent knowledge for this action is given in Figure 4.12. What information about this action must the system give to an agent in order for the agent to know how to do the intended action? The private knowledge that the system has for opening the panel is shown in Figure 4.13. If the agent is advanced (U3), then the information that the action has the post-condition of the panel being open is sufficient. That is, in the theorem-prover, the following query

#### U3(postcondition(a1,configuration(panel1,state(open))) -> howToDo(a1))

can be proven since the advanced agent's knowledge (including knowledge inherited from lower agent types) contains the knowledge that the post-condition information implies the rest of the information about the action and its sub-actions, i.e. concrete(a1) and termination(a1) can be proven.

For beginner agents (U2), the post-condition knowledge alone does not provide enough information; that is, a query similar to the one above (replacing U3 with U2) could not be proven. However, providing the information that removing the screw (i.e. taking the screw out of the hole) is a sub-action of opening the panel allows the beginner agent to figure out the rest of the action information. The query that succeeds in this case is:

<sup>&</sup>lt;sup>11</sup>For reasons explained in the next chapter, this rule is not actually used in the generation process.

```
% Action: a1
S during(a1, t_1).
S starts(t_1, t).
C result(a1, t_2).
S meets(t_2, t_1).
S agent(a1, u).
S postcondition(a1, configuration(panel1, state(open))).
S subactions(a1, oa1).
S nextAction(a1, a2).
S parent(a1, a).
% Action: oa1
S during(oa1, t_1_1).
S starts(t_1_1, t_1).
C result(oa1, t_1_2).
S meets(t_{1_2}, t_{1_1}).
S agent(oa1, u).
S postcondition(oa1, configuration(screw1, location(awayFrom(hole1)))).
S subactions(oa1, ua1).
S nextAction(oa1, oa2).
S parent(oa1, a1).
% Action: oa2
S during(oa2, t_1_2).
S finishes(t_{12}, t_{1}).
C result(oa2, t_2).
S agent(oa2, u).
S precondition(oa2, oa1).
S postcondition(oa2, configuration(cover1, location(awayFrom(panel1)))).
S nextAction(oa2, nil).
S prevAction(oa2, oa1).
S parent(oa2, a1).
```

Figure 4.13: Action information for opening the panel (and its sub-actions)

```
% Action: ual
S during(ua1, t_1_1).
S starts(t_1_1_1, t_1_1).
C result(ua1, t_1_2).
S meets(t_1_1_2, t_1_1_1).
S agent(ua1, u).
S instrument(ual, screwdriver).
S motion(ua1, screw1, rotational).
S pathDir(ua1, ccw).
S postcondition(ua1, configuration(screw1, state(loose))).
S nextAction(ua1, ua2).
S prevAction(ua1,nil).
S parent(ua1, oa1).
% Action: ua2
S during (ua2, t_1_1_2).
S finishes(t_{1}_{2}, t_{1}_{1}).
C result(ua2, t_1_2).
S agent(ua2, u).
S motion(ua2, screw1, translational).
S pathEnd(ua2, awayFrom(hole1)).
S nextAction(ua2, nil).
S prevAction(ua2, ua1).
S parent(ua2, oa1).
```

Figure 4.14: Action information for opening the panel (cont'd)

```
U2(postcondition(a1,configuration(panel1,state(open))),
    subactions(a1,oa1),
    postcondition(oa1,configuration(screw1,location(awayFrom(hole1))))
    -> howToDo(a1)).
```

indicating the information the beginner agents need in order to perform the intended action. Finally, for novice agents (U1), all of the action information (except its instrument) must be provided in order for the agent to perform the action.

Thus, varying amounts of action information is needed by the different agent types in order to carry out actions as intended. By using modal operators when predicating agent knowledge, different levels of experience with domain can be represented and reasoned about in order provide the appropriate action information to the agents.

### 4.5 Conclusion

Knowledge needed to reason about the domain falls into one of the types discussed above (object, action, and agent). With the mFOL representation that SPUD uses, any other knowledge needed to model other aspects of the domain, or completely different domains, could be added straightforwardly. Each of these types of information has been studied previously for various reasoning tasks and the mFOL representation can be used for the same reasoning tasks. However, these sources of information allow SPUD to reason about Natural Language Generation tasks as well.

The ability to represent which pieces of action information contribute to the concreteness and the termination information of an action anchors the representation of this domain. Without the ability to represent and reason about such information, it could not be determined whether actions were sufficiently defined to be carried out. Such an inability affects many reasoning tasks, such as planning. However, beyond the general usefulness of representing such information, the same representation supports reasoning about actions for Natural Language Generation purposes.

SPUD reasons about what the agent knows and then generates sentences to describe actions (and whatever else needs to be described). The resulting text relies on the agent's

knowledge and therefore it is tailored to the agent's experience with the domain. Based on the general and specific knowledge assigned to each agent expertise level, less experienced agents will receive more detailed instructions than those with more experience, who know how to do more in the domain. Without the ability to represent different agent expertise levels, such tailoring of generated text would not be possible. The power of SPUD lies in its use of the independently-motivated and powerful mFOL representation to generate such tailored Natural Language instructions efficiently.

# Chapter 5

# Generating Instructions using SPUD

Along with the domain knowledge described in the previous chapter, SPUD needs lexical information in order to generate instructions for actions in the domain. Lexical information includes syntax, semantics, and pragmatics for individual lexical items (i.e. words and phrases). Syntactic information is represented by a Feature-Based Lexicalized Tree-Adjoining Grammar (FB-LTAG), which is a powerful general-purpose lexical representation (see Section 2.6). In SPUD, however, semantics and pragmatics are also associated with lexical items, unlike in standard FB-LTAG. In addition, pragmatic information is associated with individual trees, each of which can represent the syntax of many different entries, thus specifying when it is appropriate to use a particular syntactic construction (tree). The first two sections of this chapter present the language model (i.e. the lexical items with their associated trees) used in the generation of instructions for the domain actions mentioned in Section 4.1.

SPUD employs a greedy algorithm to choose which lexical items to use in a description of an entity (object, event, or action) which must satisfy given communicative goals. SPUD uses a modal first-order logic theorem-prover [Stone, 1998] (see Section 2.3) to reason about the effect of including certain lexical items in the description, such as what the hearer infers about the entity given the description and previous knowledge. In this way, SPUD generates tailored, effective instructions based on the language model and a model of the hearer. SPUD's algorithm is described in Section 5.3 and illustrated through the generation of example instructions in Section 5.4. Section 5.5 discusses the benefits and costs of using SPUD for generating effective instructions.

# 5.1 Basic Lexical Information

As described in Section 2.6, FB-LTAG provides a formal, linguistically-sound way of specifying lexical items and their syntactic constructions and how they combine together to form larger constructions. However, in order to generate text using FB-LTAG, semantic and pragmatic information must be associated with entries in the grammar. In SPUD, semantic information appears with individual lexical items and pragmatic information can appear with lexical items and/or their associated syntactic constructions.

This work does not rely heavily on the syntactic details of TAG. The most important point is how the semantics is built along with the syntactic tree. As described in [Stone and Doran, 1997], when a substitution or adjunction operation is applied to a tree, the semantics of the substituted or adjoined tree is simply *conjoined* to the semantics of the original tree. Therefore, the semantics of a complicated syntactic construction is easy to compute. The operation of SPUD relies on this ease of computation, as is shown in Section 5.3.

The next two sub-sections describe how lexical items and trees are specified in SPUD. The remaining sub-sections briefly describes the basic lexical items of the implemented language model.

#### 5.1.1 Specifying lexical items

Each lexical item in the grammar includes the word(s) which serve as anchors for the TAG trees, the type of node that it represents, features to match when considering using it at a certain node in a partial tree, a list specifying the trees that it anchors, its complete semantics (separated into what is *asserted* and what is *presupposed* by the lexical item), and its pragmatics. An example lexical item, giving information for the transitive verb

```
word = { name = { turn }
    basic = { true }
    decl = { alpha(S,R,E,O) }
    site = { vp(S,R,E) }
    match = { () }
    semantics = { during(E, R), motion(E, 0, rotational) }
    presupposition = { true }
    pragmatics = { property(0, turnable, true) }
    trees = { transitiveVP(S,R,E,O) } }.
```

Figure 5.1: Lexical item entry for the verb turn

turn, is shown in Figure 5.1. The fields in a lexical item entry are as follows:

- name gives the lexical item.
- basic indicates whether the lexical item is considered a basic word, in the sense of [Rosch, 1978].
- decl indicates the type of trees<sup>1</sup> that the lexical item anchors and a list of semantic indices (i.e. arguments) used in the trees.
- site indicates the type of node which the lexical item can expand, needed for determining which lexical items are applicable to a given partial tree. This field indicates the top node of the trees associated with the lexical item (see below). While this could be automatically extracted from the associated trees, listing it explicitly with the lexical item is more efficient, similar to declaring a function's prototype in a computer program.
- match gives syntactic features on the site node which must be matched in order to use the lexical item at that site.
- semantics contains the meaning asserted by the lexical item, using the same predicates as in the domain and action knowledge representation. Existential quantifiers

<sup>&</sup>lt;sup>1</sup>alpha indicates initial trees and beta auxiliary trees; see Section 2.6.

(e.g., Q for  $\exists Q$ ) can be used to predicate the existence of some entity which does not appear as a semantic argument in the trees of the lexical item (see the lexical item *remove* in Figure 5.7).

- presupposition indicates what must be shared knowledge in order for the lexical item to be selected. See the lexical item *open* in Figure 5.7 for an example. This is a useful feature for producing efficient texts [Stone and Webber, 1998], but is not used to its full advantage in this implementation.
- pragmatics provides constraints on the situations in which the lexical item can used. For instance, *turn* can only be used when its object has the property of being turnable.<sup>2</sup> In Figure 5.9, the pragmatics command for the empty subject lexical item indicate that the subject can only be omitted if commands (i.e. imperative sentences) are being generated. If the pragmatics for a lexical item does not hold, the lexical item will not be considered for addition to a partial tree.
- trees gives the names of those trees which the lexical item can anchor. The full structure of these trees is provided separately (see the next section), as many lexical items can anchor the same tree.

In the presentation of particular lexical items, only those fields with relevant information are given; the decl field and fields which are empty or only contain true are omitted.

#### 5.1.2 Specifying syntax

Each elementary tree used in SPUD has associated with it a complex name which includes its arguments, relevant pragmatic information, and a tree structure. A lexical item can participate in many syntactic constructions, some of which may be inappropriate in a particular situation. Thus, pragmatic information indicates when it is appropriate to use the syntactic structure.

Tree structures specify information for each node in the tree, including their type, their features (top and bottom), whether they are substitution sites (subst) or adjunction sites

<sup>&</sup>lt;sup>2</sup>While the argument can be made that this is a semantic constraint that would be better placed in the presupposition of lexical items (see [Stone and Doran, 1997]), I have consistently included such constraints in the pragmatics of lexical items of the implemented language model.

```
entry = {
  name = { simpleS(S,R,E,A) }
  pragmatics = { present(R) }
  tree = {
    node = \{
      type = { s(S,R,E,A) }
      top = { (cat s) }
      bottom = { (cat s) }
      kids = {
        subst = {
          type = { u:np(A) }
          top = { (cat np; number X; person Y; case nom) }
        }
        node = \{
          type = { ip(S,R,E) }
          top = { (cat ip; form main; number X; person Y) }
          bottom = { (cat ip; form main; number X; person Y) }
          kids = {
            node = \{
              type = { u:infl(S,R,E) }
              top = { (tense present; form main; number X; person Y) }
              bottom = { (tense present; form main; number X; person Y) }
              kids = { anchor = { index = \{1\} \} }
            }
            subst = {
              type = { vp(S,R,E) }
              top = { (cat vp; tense present; form main; number X; person Y) }
            }
      }
}
    }
   }
 }
}.
```

Figure 5.2: Full SPUD specification of the simpleS tree



Figure 5.3: Graphical specification of the simpleS tree

(foot), and the nodes which are their children (kids). Figure 5.2 shows the full tree entry for the basic sentence structure used in this implementation. Features associated with nodes pass information between nodes, such as tense and agreement information. Since features are not vital to this dissertation,<sup>3</sup>, trees are shown graphically (as in Figure 5.3) throughout the rest of this chapter.<sup>4</sup> (Appendix A.2 contains the full specification of all of the tree entries.)

One final note about the tree specification. The u: appended to a node name indicates that the information provided by that node need not be new to the hearer. By specifying that a node need not contain new information, SPUD can choose to add lexical items which would otherwise not meet its criteria. Although this implementation has trees which use u: for some nodes, this is not a vital aspect of the implementation and is more for further work which might involve anaphors (e.g. pronouns).

#### 5.1.3 Discourse and sentence structure

The top-level "lexical items" are those which represent sentence boundaries in a discourse segment (i.e. instruction step). The anchors for these trees include punctuation, specifically the period. The basic discourse segment consists of a single sentence followed by a period,

<sup>&</sup>lt;sup>3</sup>The only two features used in this implementation which affect the final output are the *tense* and *form* features. Tense is needed by the morphological processing to inflect the verb correctly. Form is needed by the "means clause" entry (see Section 5.2.2) where the subordinate verb phrase must be in gerund form.

 $<sup>{}^{4}</sup>$ As note earlier,  $\diamond$  indicates the anchor,  $\downarrow$  denotes a substitution site, and \* specifies an adjunction site.

	word = {	name = { . }
Lexical item:	}.	<pre> site = { ds(S,R,E) } semantics = { step(E), during(E,R) } trees = { ds1(S,R,E) }</pre>

```
Tree entry: ds(S,R,E) = s(S,R,E) \downarrow \quad .
```

Figure 5.4: Basic discourse segment entry

as shown in Figure 5.4. Other discourse segment entries are discussed in Section 5.2.

The next level consists of sentence trees which are anchored by inflection (generally tense) which affects the form of the verb (i.e. *remove* vs *removes* vs *removed*). I have used only two types of sentences anchored by the inflection *present tense* since only imperative instructions are to be generated in this implementation. The first type of sentence corresponds to the standard rule  $S \rightarrow NP VP$  (Figure 5.5) and is used for actions or events. The second is a predicative sentence,  $S \rightarrow NP$  be Pred (Figure 5.6), used for describing states, e.g. "the screw is loose".

#### 5.1.4 Verb phrases

Only one type of verb phrase, namely transitive, is involved in the example instructions (besides the predicative "be" verb phrases). Transitive verb phrases consist of the verb and its direct object (see Figure 5.7). One reason for not addressing other types of verb phrases, such as intransitive or ditransitive, is the nature of maintenance instructions. They involve the manipulation of objects, thus intransitive verb phrases (i.e. verb phrases without direct objects) are not found. The explicit manipulation of more than one object at a time or the involvement of another agent is rare in these instructions, so ditransitive verb phrases (i.e. verb phrases with a direct object and a *required* indirect object) are



Figure 5.5: Simple sentence entry



Figure 5.6: Predicative sentence entry

| be also rare. However, maintenance instructions frequently involve the motion of objects and therefore prepositional phrases are needed to specify path information for the actions involving motion. Such path information is optional and therefore not considered part of the verb's argument structure. To give SPUD the option to include path information in an action description, prepositional phrases are encoded as auxiliary trees which adjoin onto verb phrases. The lexical item to introducing a prepositional phrase is used to indicate the end of a path (see Figure 5.8).<sup>5</sup> (For a discussion of encoding motion verbs and their properties using a TAG formalism, see [Palmer *et al.*, 1999].)

#### 5.1.5 Noun phrases

Two types of regular noun phrases are encoded in the current implementation, an indefinite noun phrase with the determiner *a* and a definite noun phrase without a determiner (since it is typically omitted in maintenance instructions).<sup>6</sup> Pragmatics, in this case, whether the object is uniquely identifiable or not, determines which NP tree is used.<sup>7</sup> The semantics for nouns consists of its type, e.g. type(0,panel). Any additional information about an object, such as a label, can be adjoined into a noun phrase with an adjective auxiliary tree. For the definition of these lexical items and trees, see Appendix A.

In addition to these regular noun phrases, a noun phrase for an empty subject is specified so that proper subject-less (i.e. imperative) instructions can be generated. The lexical item for the empty subject (indicated by the specification for *nominative* case in the **match** field) has the semantics that the subject is the hearer and pragmatic information which indicates that it is only appropriate when the predicate **command** holds, indicating the generation of instructions (see Figure 5.9).<sup>8</sup> A similar tree structure (with different syntactic features) could be used to generate instructions which elide the object which is in focus, a common occurrence in naturally-occurring instructions (e.g. *Slide valve aft and* 

<sup>&</sup>lt;sup>5</sup>The predicate locObj is used to take the object out of a location relation; for example, locObj(at(panel1),panel1)

holds and can be used to bind a variable (such as 0) to the object identifier (panel1).

<sup>&</sup>lt;sup>6</sup>While plural noun phrases appear frequently in maintenance instructions, they are not addressed in this dissertation. While the number of the noun phrase (i.e. singular vs. plural) which serves as a verb's argument does affect the termination information and other performance information of the action, it is not one of the sources of termination information that I chose to address.

<sup>&</sup>lt;sup>7</sup>As implemented, all objects in the domain are uniquely identifiable.

<sup>&</sup>lt;sup>8</sup>The e which serves as the anchor for the tree is removed by morphological processing.

#### Lexical entries:

```
word = {
                    \{ open \}
name =
. . .
                    \{ vp(S,R,E) \}
site =
. . .
                    { during(E,R),
semantics =
                    postcondition(E,configuration(0,state(open))) }
                    { configuration(R,0,state(closed)) }
presupposition =
. . .
                    { transitiveVP(S,R,E,O) }
trees =
}.
word = {
                    { move }
name =
. . .
                    \{ vp(S,R,E) \}
site =
. . .
semantics =
                    { during(E,R), motion(E, 0, translational) }
. . .
pragmatics =
                    { property(0, movable, true) }
                    { transitiveVP(S,R,E,0) }
trees =
}.
word =
       ł
                    { remove }
name =
. . .
                    \{ vp(S,R,E) \}
site =
. . .
                    { during(E,R),
semantics =
                    (?L?X configuration(R,O,location(L)), locObj(L,X),
                    postcondition(E,configuration(0,location(awayFrom(X))))) }
. . .
                      property(0, movable, true) }
pragmatics =
                    {
trees =
                    { transitiveVP(S,R,E,O) }
```

Tree entry:  
$$v(E)$$
  $u:np(O)$   
 $\downarrow$ 

Figure 5.7: Transitive verb entries for open, remove, and turn









Figure 5.9: Lexical entry for an empty subject

*remove*). This implementation does not address such eliding of objects in instructions, but SPUD is capable of handling such a construction.

# 5.2 Multi-clausal lexical items

In order to express necessary action information such as sub-actions, some instructions require multi-clausal sentences or even multiple sentences. The multi-clausal lexical items which make these instructions possible are described in this section. In order to choose between a single clause and multiple clauses, however, SPUD must be given preferences to guide the choice of the appropriate lexical items. Since SPUD uses a greedy algorithm without backtracking (see Section 5.3), such guidance is crucial. If SPUD chooses a lexical item which precludes improvements later in the generation process, SPUD has no way to undo the choice. As currently implemented, SPUD must make the right choice first. Thus, the issue of stylistic preferences, i.e. guiding SPUD to make the right choice, is presented before the presentation of the multi-clausal lexical items.

#### 5.2.1 Stylistic preferences

Many researchers have addressed the issue of generating tailored text to reflect stylistic preferences when addressing different types of hearers. (See the discussions of [Paris, 1988; Bateman and Paris, 1989; McKeown *et al.*, 1993; Nicolov *et al.*, 1996] in the next chapter.) [Bateman and Paris, 1989], in particular, recognize that hearers with different levels of experience with the domain require different phrasing of information. When implementing lexical constructions for this dissertation, I found that in order to get appropriate phrasings and levels of detail based on the agent type, SPUD needs either information about agents in addition to the agent knowledge discussed in Chapter 4. Therefore, as part of agent knowledge, I include how agents view certain kinds of actions, which affects how SPUD describes them to different agents.

Since an agent's view of actions is private to the agent and should not be treated as inheritable or shared by agents at other levels, the nested modal operators described in the previous chapter will not work. Additional modal operators are needed to predicate knowledge of the agent types' views of actions. They are defined as follows:

- **P1** encompasses the novice agent type's views of actions and knowledge of the domain. In terms of the other modal operators, it inherits all the knowledge predicated with and inherited by **U1**, i.e.  $P1 \supset U1$ .
- **P2** represents the beginner agent type's views and knowledge, such that  $P2 \supset U2$ . Notice that knowledge predicated with **P1** is not included in the knowledge of the beginner agents (unlike knowledge predicated with **U1**).

**P3** represents the advanced agent type's views and knowledge, such that  $P3 \supset U3$ .

For the system to use knowledge of agent type in generating instructions, its knowledge must encompass all of the assumed knowledge of the agents, including agents' views of actions. That is,  $\mathbf{S} \supset (\mathbf{P1} \cup \mathbf{P2} \cup \mathbf{P3})$  where  $\mathbf{S}$  is the system's modal operator. With this modal machinery in place, preferences (i.e. views) for each agent type can be defined.

For novice agents, non-basic (i.e. higher-level) actions in the domain seem complicated since they involve sub-actions. This view is encoded in a predicate called **complex**. For instance, the fact that a novice agent views opening any object in the domain as a complicated action is represented by the following rule:<sup>9</sup>

\*A \*O (postcondition(A,configuration(O,state(open))) -> P1 complex(A)).

Some particularly complicated higher-level actions, which involve complex sub-actions, require a separate predicate called **elaborate** to indicate how novice agents view such actions. For beginner agents, certain actions are viewed as purposes for other actions, indicating beginner agents' increased understanding of the domain. A **purpose** predicate is used to indicate those actions which are viewed as purpose actions.

In the following presentation of multi-clausal lexical items, these predicates appear in the pragmatics for some of the lexical item entries, indicating when particular constructions should be used because of the agents' view of an action.

 $<sup>^{9}</sup>$ Alternatively, rules which reason about the hierarchical level of the action could be used instead of relying on semantic features of the action.





Figure 5.10: Means clause entry

#### 5.2.2 Purpose and means clauses

One way of expressing how to do an action is to describe the means by which it can be accomplished. This can be done by adjoining to the main clause a subordinate *means* clause introduced by the lexical item *by*. The *means clause* lexical item and tree is shown in Figure 5.10.<sup>10</sup> This is the simplest way to express an action's sub-actions.

Sometimes, however, it may be appropriate to emphasize a sub-action by putting it in the main clause. Sentences with *purpose clauses* achieve this since the subordinate action is described in the main clause and the purpose is described in a subordinate clause anchored by *to*. The two types of purpose clauses, discussed in Section 3.3.3, differ in whether the purpose clause expressing the higher-level action is before or after the main clause. The case where the purpose clause is adjoined to the end of the main clause is shown in Figure 5.12. Due to the hierarchical nature of actions in maintenance tasks and the fact that the

<sup>&</sup>lt;sup>10</sup>beta in the decl field indicates that the lexical item is associated with auxiliary trees which are added to other trees using adjunction.

	word = {	name =	{ . }
		 sito =	{ de(S R F) }
		5100	
Lexical item:		semantics =	<pre>{ step(E), during(E,R), substep(P,E), during(P,T) }</pre>
		 pragmatics =	<pre>{ purpose(E) }</pre>
	}.	trees =	{ ds3(S,T,P,R,E) }
Tree entry:		ds(S,R,E)	
	s(S,T)	$(\mathbf{P}, \mathbf{P}, \mathbf{A}) \downarrow \diamond$	

Figure 5.11: Discourse segment entry for emphasizing actions done for a purpose

higher action is typically the one being described, a discourse segment tree encodes the emphasis on the sub-action (i.e. its expression in the main clause). In the normal discourse segment entry (see Figure 5.4), the sentence node is given as s(S,R,E,A), where E is the higher action. However, in the discourse segment for cases in which the sub-action is done for the purpose of the higher-level action, the sentence node is s(S,T,P,A) where P is the subordinate action and T is the time interval in which it occurs (see Figure 5.11). The pragmatics of this discourse segment tree indicates that if the agent views the main action as a purpose action, then the discourse segment entry shown in Figure 5.11 will be used.

Once this discourse segment entry is chosen based on its pragmatics, the lexical item to can adjoin a purpose clause to express the high-level action.

When an action is viewed as **complex** by the hearer, I want to be able to express the main action in a fronted purpose clause (see Figure 5.13) in order to provide a framework for interpreting the subordinate action described in the main clause (see Section 3.3.3). Unlike the non-fronted purpose clause which adjoins to a verb phrase, this one must adjoin onto a sentence node. This allows complicated descriptions of the sub-action(s) to be in multiple clauses in the sentence without creating a convoluted sentence. The pragmatics of this construction specifies that in order for this fronted purpose clause to be applicable,



Figure 5.12: Purpose clause entry

SPUD must be generating instructions (i.e. command  $holds^{11}$ ) in addition to the main action being viewed as complex by the agent.

#### 5.2.3 Until clauses

Some maintenance instructions use *until* clauses to convey an action's post-condition. In SPUD, these adjoin onto verb phrases, providing a substitution site for a sentence describing the post-condition of the action (see Figure 5.14).<sup>12</sup> (A similar tree would be used by other lexical items, such as *while*, which involve the adjunction of a sentence to a verb phrase, but different features would need to be associated with the adjoined sentence to reflect the different syntactic requirements of the lexical item.) Since an action's result time interval is common knowledge, this knowledge can be presupposed semantics of the lexical item.

<sup>&</sup>lt;sup>11</sup>Alternatively, if sentence type were included in the features of the trees, the match field in the lexical item could be used to enforce the need for an imperative sentence.

<sup>&</sup>lt;sup>12</sup> "Fronted" *until* clauses, such as in "Until the light turns green, hold the lever at the RESET position", are not found in the corpus of maintenance instructions described in Chapter 3.



Figure 5.13: Fronted purpose clause entry



Figure 5.14: Until clause entry



Figure 5.15: Entry for conjoining verb phrases ("and then")

#### 5.2.4 Conjoining clauses

For instructions that need to describe two consecutive actions in the same sentence, the tree anchored by "and then" in Figure 5.15 is used to adjoin another verb phrase to an existing verb phrase. Although there can be problems with conjunction due to infinite recursion, this entry has been used without difficulty for the example instructions.

#### 5.2.5 Multi-sentence instruction steps

Complicated actions with multiple sub-actions require multiple sentences to describe. A treatment similar to that of adjoining verb phrases as in Figure 5.15 would be preferable for issues of uniformity and discourse structure. However, in the interest of efficiency for this implementation, the entry for a multi-sentence discourse segment shown in Figure 5.16 is used. It is anchored by periods and the word *then* (to indicate the sequential nature of the sub-actions), with the pragmatics of elaborate(A), indicating that this discourse segment structure should be used when the agent can be assumed to view the high-level action as very complicated. Without the statement of its pragmatics and the accompanying agent knowledge bearing out the appropriateness of this tree, the use of this tree would not be given a high enough ranking to be chosen as the discourse segment structure since it has



Figure 5.16: Multi-sentence discourse segment entry

more unfilled substitution sites than other choices. The next section discusses SPUD's algorithm for generating text, which should clarify the need for such stylistic preferences.

# 5.3 The SPUD Algorithm

When told to describe a particular action instance, SPUD uses the information about the action, the agent, the domain, and the lexical items to choose a lexical item which best furthers the description of the action and the satisfaction of other communicative goals. It employs a simple greedy algorithm, briefly described in Figure 5.17.<sup>13</sup> (See [Stone, 1998] for a full description.)

One communicative goal that SPUD always has is to identify all entities in the description uniquely. SPUD calculates the *distractors* for an entity given the information in its description so far in order to figure out whether the entity is uniquely identifiable from the description. Distractors are those entities in the *domain* of the entity that can have

<sup>&</sup>lt;sup>13</sup>A possible variant of this algorithm is one which relaxes the restriction of requiring additions to provide immediate improvement. Additions which do not improve the tree immediately can pave the way for future additions which will satisfy goals.

- Start with a tree with one node, which is a substitution site for a given type (e.g. DS or NP), to describe a given entity (e.g. action or object).
- While the current tree contains unfilled substitution sites or there are unsatisfied goals:
  - Consider all trees resulting from a single addition (i.e., a substitution or adjunction) to the current tree.
  - Compute the rank of the resulting trees based on
    - \* the number of goals satisfied,
    - \* the number of distractors for the unsatisfied goals,
    - \* the number of flaws (e.g., unfilled substitution sites),
    - \* the specificity of licensing (semantic) information (i.e., SPUD gives a lower rank to trees which provide a subset of the semantic information provided by another tree), and
    - \* whether the added lexical item is basic or not.
  - If there are no lexical items which can be added to the tree or there is no improvement in satisfying goals, leave the loop.
  - Otherwise, make the highest ranking tree the current tree and go to the beginning of the loop.
- Return the current tree (after morphological processing) and its derivation status:
  - If it satisfies all goals, then SPUD reports derivation completed successfully.
  - If it satisfies some goals but none of the possible additions could satisfy unsatisfied goals, then SPUD reports no more improvement.
  - If there are unsatisfied goals and unfilled substitution sites, but no lexical items which could be added at all, then SPUD reports *no actions possible*.

Figure 5.17: SPUD's algorithm

the same description. In order to allow SPUD to reason whether an entity is uniquely described, the knowledge provided to SPUD must include **domain** statements for all entities that need to be described. The following rules suffice for providing these statements for objects and locations, indicating that any object could be confused with any other object and similarly for locations:

```
*01 *02 (object(01), object(02) -> C domain(01,02)).
*P *Q (location(P), location(Q) -> C domain(P, Q)).
```

In addition, each action instance needs to have domain statements as well; however, in this implementation, action entities (i.e. instances) have only one domain statement, namely that it is the only entity its domain (e.g. C domain(a1,a1)). If the calculated distractor list for an entity is not empty, then further information must be included in its description to identify it uniquely.

Other types of communicative goals, such as making sure an action's description is concrete and includes termination information, are also consulted when deciding which lexical item to add to a tree. SPUD considers all of the information provided by the tree that it has built so far. If a communicative goal has not been satisfied, then a lexical item which provides information which will satisfy the communicative goal will be highly ranked for inclusion in the description. Of value to the current work is that SPUD's algorithm accounts for the fact that termination information can be provided by many different parts of a sentence. If SPUD is given the communicative goal of conveying termination information and the current tree does not yet provide it, SPUD will try to add a lexical item (possibly introducing another clause) which gives termination information. In the next section, the generation of example instructions is described in detail, illustrating how SPUD's algorithm works.

## 5.4 Generating Example Instructions

A set of three higher-level actions makes up the actions considered for this dissertation. These are sub-steps of one major action in the domain, that of restarting the pump once it has halted because of high pressure. (See Section 4.1 for the description of the domain and



Figure 5.18: Actions for example instructions

Section 4.3.1 for the action representation.) Each of the three actions have sub-actions of its own, as shown in Figure 5.18. The leaves of this action/sub-action tree represent those actions considered basic, i.e. those with no further breakdown into sub-actions.

In Section 5.4.2, I describe in detail how the instructions for the first action are generated for each of the agent types. The description pulls together how all of the information discussed in this chapter and the previous one are used by SPUD to generate effective instructions. In Sections 5.4.3 and 5.4.4, I point out interesting features of the generation of the instructions for the other two actions. First, however, SPUD needs one more piece of information in order to generate the instructions.
```
gen = {
  name = { Step 1 for Advanced }
  private = { S $ }
  shared = { P3 (P3 present(t_1) -> $) }
  describe = { ds(s,t_1,a1) }
  pattern = { ds(S,R,E) }
  features = { () }
  communicate = { concrete(a1) termination(a1) }
}.
```

Figure 5.19: An example generation instance (Step 1 for Advanced)

### 5.4.1 Generation instances

As can be surmised from the discussion of SPUD's algorithm, SPUD needs to know the entity to be described, the type of syntactic category (i.e. DS, S, etc.) to describe the entity, communicative goals to be achieved by the description of the entity, and the specification of the modal operators for the system's knowledge and the agent's knowledge. These pieces of information are provided to SPUD in a *generation instance* which is used to start the generation process. Figure 5.19 shows an example generation instance. The fields that make up a generation instance are as follows:

name states the name to be displayed in the list of defined generation instances.

- private gives the modal operator, e.g. S, designated for the system's private knowledge;
   \$ stands for an arbitrary statement, indicating that any statement predicated with
   or inherited by the designated modal operator is system knowledge.
- shared gives the modal operator, P3, designated for the hearer's assumed knowledge. An additional piece of information, namely P3 present(t\_1), is added to the assumed knowledge using the form shown in the example. This indicates that during the generation process, present(t\_1) should be considered shared knowledge thus licensing the use of present tense lexical items.<sup>14</sup>

<sup>&</sup>lt;sup>14</sup>Another way to accomplish this would be to use features field.

- describe indicates the type of syntactic construction to be generated; this includes the specification of the entity (and related entities), in the form of the arguments of the syntactic construction, to be described; in this case, a discourse segment is specified with the arguments: s (the speech time), t\_1 (the event time), and a1 (the event/action).
- pattern indicates the argument structure of the syntactic construction indicated in the describe field; it is used to match against the site field in lexical items.
- **features** lists any features required of the syntactic construction; this field has not been used in any of the generation instances in this dissertation.<sup>15</sup>
- **communicate** lists communicative goals (in addition to that of describing the entity uniquely) to be achieved by the generated text; e.g. the statements concrete(a1) and termination(a1) serve as communicative goals that must be satisfied, i.e. the generated text must supply information to make these statements provable using the shared modal operator.

### 5.4.2 Opening the panel

Opening the panel is the first step in the sequence of actions for re-starting the pump (see Figure 4.2). The action information for this top-level action, called a1, is shown in Figure 5.20. The step predicate is used to indicate the action's standing in the action/sub-action tree; in this case, it is a main instruction step and therefore is suitable to be described using a discourse segment (see Section 5.1.3).<sup>16</sup> The rest of the action information conforms to the action representation presented in Section 4.3.1. The action occurs during the time interval t\_1, which starts the time interval t, and results in the time interval t\_2. The agent of the action is the hearer, represented by u as indicated by the following statements included in the domain knowledge:

<sup>&</sup>lt;sup>15</sup>The **features** field could be used to require the generated text to be in present tense, instead of using the additional piece of knowledge in the **shared** field. If it were to be used in this way, the **features** field for the example would be (tense present).

<sup>&</sup>lt;sup>16</sup>A rule could be used to specify this information automatically by checking the distance from the action to a leaf below it or to the top-most action.

```
S step(a1).
S during(a1, t_1).
S starts(t_1, t).
C result(a1, t_2).
S agent(a1, u).
S postcondition(a1, configuration(panel1, state(open))).
S subactions(a1, oa1).
S nextAction(a1, a2).
S parent(a1, a).
```

Figure 5.20: Action information for opening the panel

C hearer(u).

C domain(u,u).

The action's post-condition states that the panel is in the state open and the action has oa1 as its first sub-action.<sup>17</sup> Finally, the action that follows a1 is a2 and its parent is the action named a.<sup>18</sup>

The action information for a1's two sub-actions, called oa1 and oa2, is shown in Figure 5.21. The first sub-action, oa1, has the additional property of being a, which is used in the semantics of the lexical items for discourse segments. Otherwise, the action information is self-explanatory. The first sub-action has sub-actions of its own, shown in Figure 5.22. These detail the steps involved in removing the screw: loosening the screw and taking it out of the hole. Given these three levels of action detail, the action description for a1 can have many forms depending on the agent's expertise and the communicative goals to be satisfied.

The lexical items that could be needed for this action description include: verbs for *opening*, *removing*, and *turning*; nouns for *panel*, *cover*, and *screw*; an adverb for *counter-clockwise*; *until*; purpose and means clauses (*to* and *by*); the empty subject noun phrase for imperative; sentence and discourse segment entries. Specification of these lexical items can be found in previous sections or Appendix A.

<sup>&</sup>lt;sup>17</sup>Only the first sub-action is listed since nextAction statements are used to find the next sub-action.

<sup>&</sup>lt;sup>18</sup>An additional statement about the action is the domain statement, i.e. C domain(a1,a1), discussed previously.

```
% Action: oa1
S substep(oa1, a1).
S during (oa1, t_1_1).
S starts(t_1_1, t_1).
C result(oa1, t_1_2).
S agent(oa1, u).
S postcondition(oa1, configuration(screw1, location(awayFrom(hole1)))).
S subactions(oa1, ua1).
S nextAction(oa1, oa2).
S prevAction(oa1, nil).
S parent(oa1, a1).
% Action: oa2
S during (oa2, t_1_2).
S meets(t_{1_2}, t_{1_1}).
S finishes(t_1_2, t_1).
C result(oa2, t_2).
S agent(oa2, u).
S precondition(oa2, configuration(screw1, location(awayFrom(hole1)))).
S postcondition(oa2, configuration(cover1, location(awayFrom(panel1)))).
S nextAction(oa2, nil).
S prevAction(oa2, oa1).
S parent(oa2, a1).
```

Figure 5.21: Action information for removing the screw and the panel

```
% Action: ua1
S during(ua1, t_1_1).
S starts(t_1_1_1, t_1_1).
C result(ua1, t_1_2).
S agent(ua1, u).
S instrument(ual, screwdriver).
S motion(ua1, screw1, rotational).
S pathDir(ua1, ccw).
S postcondition(ua1, configuration(screw1, state(loose))).
S nextAction(ua1, ua2).
S prevAction(ua1,nil).
S parent(ua1, oa1).
% Action: ua2
S during(ua2, t_1_2).
S meets(t_1_1_2, t_1_1_1).
S finishes(t_1_1_2, t_1_1).
C result(ua2, t_1_2).
S agent(ua2, u).
S motion(ua2, screw1, translational).
S pathEnd(ua2, awayFrom(hole1)).
S nextAction(ua2, nil).
S prevAction(ua2, ua1).
S parent(ua2, oa1).
```

Figure 5.22: Action information for how to remove the screw

Figure 5.23: Agent knowledge for Step 1

The assumed knowledge of the different agent types with respect to this action is shown in Figure 5.23. These rules state that:

- advanced agents know that opening the panel involves the sub-action of removing the screw;<sup>19</sup>
- beginner agents know that removing the screw involves a set of sub-actions for loosening the screw and taking it out of the hole;
- novice agents know that opening a panel involves taking away its cover and that turning a screw involves a screwdriver;
- and finally, an action involving the opening of any object should be considered a complex action for a novice agent.

Using this agent knowledge, SPUD can determine the most appropriate action description for a1 when addressing a specific agent type.

<sup>&</sup>lt;sup>19</sup>Even though advanced agents are at the top-level of agents, and thus none of the lower agent levels would ever inherit knowledge from advanced agent knowledge, the potentially inheritable modal operator U3



step(a1), agent(a1,u), during(a1,t\_1), present(t\_1)

Figure 5.24: Generation of Step 1 for Advanced, part 1

The generation instance to get SPUD to generate an action description for a1 for advanced agents was given in Figure 5.19. It instructs SPUD to describe the action a1 at time  $t_1$  in a discourse segment so that the communicative goals concrete(a1) and termination(a1) are satisfied. The generation process begins with a tree with a single substitution site,  $ds(s,t_1,a1)$ . The only applicable discourse segment tree is ds1 (see Figure 5.4), with a single substitution site for a sentence,  $s(s,t_1,a1,u)$ . All the other discourse segment trees have pragmatics which do not hold and therefore are not applicable. Next, a tree which applies to a sentence node must be selected. In this case, present( $t_1$ ) holds, so the simpleS tree (see Figure 5.5) is chosen and added to the tree. At this point, the current tree and its associated semantics are as shown in Figure 5.24.

Next the verb *open* is chosen, since it furthers the description the most and satisfies the communicative goals. Once the verb phrase has been substituted, only the noun phrases remain to be filled in. The empty subject noun phrase is chosen because of its pragmatics and a definite noun phrase (without a determiner; see Section 5.1.5) is substituted for the object noun phrase. The resulting tree and its semantics are shown in Figure 5.25. Figure 5.26 shows the actual output window displayed by SPUD, indicating the order in which

is used instead of the private P3 modal operator in order to indicate that the predicated knowledge should be considered objective (as opposed to subjective and therefore private) knowledge about the domain.



Figure 5.25: Generation of Step 1 for Advanced, part 2

lexical items were added to the tree and the morphological processing that takes place (in particular, the *e* and *present* "words" are erased).

The generation instance for the beginner agent type is shown in Figure 5.27. It is the same as for the advanced agent type, except the shared modal operator information reflects the agent type. The start of the generation process proceeds the same as in the generation for the advanced agents, through choosing the verb. However, the semantics of *open* does not satisfy the concreteness communicative goal, thus SPUD looks to add another lexical item. In this case, it chooses to adjoin a means clause to describe how to do the high-level action. The means clause (see Figure 5.10) adds the sub-action information needed to satisfy the concreteness goal. After adjoining the means clause, the partial tree and its semantics are as shown in Figure 5.28. The verb *remove* is added to the tree and the noun phrases are filled in to finish the tree, resulting in the tree and semantics in Figure 5.29. SPUD's output is shown in Figure 5.30; morphological processing changes *remove* into its



Figure 5.26: Output: Step 1 for Advanced

```
gen = {
   name = { Step 1 for Beginner }
   private = { S $ }
   shared = { P2 (P2 present(t_1) -> $) }
   describe = { ds(s,t_1,a1) }
   pattern = { ds(S,R,E) }
   features = { () }
   communicate = { concrete(a1) termination(a1) }
}.
```

Figure 5.27: Generation instance: Step 1 for Beginner



Figure 5.28: Generation of Step 1 for Beginner, part 1

gerund form *removing*.

The generation instance for Step 1 (see Figure 5.31) for novice agents includes additional communicative goals, reflecting the increased detail required in order for the agents to be able to carry out the instruction. One type of communicative goal which has not been mentioned before is one which indicates the action's relations to other actions. In this case, the communicative goal nextAction(oa1,oa2) ensures that the agent will know the existence and ordering of the sub-actions. The concrete and termination communicative goals are included for actions at all levels, so that the generated instructions will provide the agent with enough detail to perform the action.



Figure 5.29: Generation of Step 1 for Beginner, part 2



Figure 5.30: Output: Step 1 for Beginner

Figure 5.31: Generation instance: Step 1 for Novice



Figure 5.32: Generation of Step 1 for Novice, part 1

The generation process starts the same as in the previous examples. However, the first lexical item chosen after the sentence tree substitution is the fronted purpose clause tree (see Figure 5.13) since its pragmatics hold, i.e. P1 complex(a1) can be proven. The view of a1 as complex by the agent means that the high-level action needs to provide a framework for interpreting the sub-actions, accomplished by the fronted purpose clause construction. Once the adjunction is done, the verb *open* is chosen for the high-level action, now in the fronted purpose clause. At this point, the partial tree and its semantics are as shown in Figure 5.32. The simpleS tree is used at the substitution site in the main



Figure 5.33: Generation of Step 1 for Novice, part 2

clause; the pragmatics of present(t\_1\_1) can be proven using the rules for reasoning about subintervals given in Section 4.4.2. Next, the verb for the sub-action, *remove*, is chosen and added to the tree, but more detail than this is needed about the sub-action. A means clause is adjoined to the verb phrase for the sub-action to indicate how to remove the screw, which a novice agent does not know how to do. After these additions, the partial tree and its semantics are as shown in Figure 5.33 (the discourse segment node of the tree is omitted due to space constraints). Next, the second sub-action is added by adjoining another verb phrase using *and then* (see Figure 5.15). For the first sub-action, path direction does not need to be added since the agent's knowledge includes the general knowledge that standard screws are turned counterclockwise. However, termination information needs to be added to the action information in the means clause, in order to satisfy the termination(ua1) communicative goal. In this case, an *until* clause is adjoined to express the sub-action's post-condition. Due to the size of the resulting tree, it is not shown in graphical format; instead, the SPUD output window is shown in Figure 5.34.

As discussed and illustrated, the generation process produces different instructions for the different agent types using the same action information. The instructions vary in the amount and presentation of action information. Agent expertise knowledge, i.e. knowledge predicated with the U1, U2, and U3 modal operators, determines which pieces of action information need to be included in, or omitted from, an action description for a particular agent type. This accounts for the varying level of detail seen in the generated instructions, summarized below ((a) for advanced, (b) for beginner, and (n) for novice):

- (24) (a) Open panel.
  - (b) Open panel by removing screw.
  - (n) To open panel, remove screw by turning screw until screw is loose and then remove cover.

Beyond the level of detail, however, the presentation style changes from agent type to agent type, depending on how the agents view the action to be done. The instruction generated for novice agents uses a fronted purpose clause to address the novices' view that opening an object is a complex action. In the discussion of the remaining steps of the task, the stylistic differences between the instructions for the different agent types will be highlighted, as will the differences in the level of detail. Much of the generation of the instructions for the second and third steps follows the same general course as that discussed in this section, and thus the following discussions do not go into the same level of detail.

### 5.4.3 Normalizing the pressure

The second step in restarting the pump is to normalize the pump's pressure, which is too high. This step is much simpler than the previous one, involving only one basic subaction. The action information for this step is shown in Figure 5.35. Notice that an explicit purpose relation exists between the main action (a2) and its sub-action (na1); this means



Figure 5.34: Output: Step 1 for Novice

```
% Action: a2
S step(a2).
S during (a2, t_2).
C result(a2, t_3).
S meets(t_3, t_2).
S agent(a2, u).
S postcondition(a2, configuration(pressure1, state(normal))).
S subactions(a2, na1).
S nextAction(a2, a3).
S prevAction(a2, a1).
S parent(a2, a).
% Action: na1
S substep(na1,a2).
S during (na1, t_2_1).
S starts(t_2_1, t_2).
S finishes(t_2_1, t_2).
C result(na1, t_3).
S agent(na1, u).
S motion(na1, dial1, rotational).
S pathDir(na1, cw).
S postcondition(na1, configuration(gauge1, state(within(range1)))).
S purpose(na1, generate(a2)).
S nextAction(na1, nil).
S prevAction(na1, nil).
S parent(na1, a2).
```

Figure 5.35: Action information for normalizing the pump's pressure

that the action of turning the dial clockwise until the pressure gauge registers normal pressure generates (directly accomplishes) the action of normalizing the pressure.

The agent knowledge for this step is shown in Figure 5.36. Advanced agents know that an action involving normalizing the pump's pressure has a sub-action with the postcondition of having the pressure gauge be in the normal range. Beginner agents know that getting the gauge within the normal range involves turning the dial in its turnable direction. Since advanced agents should also be assumed to know this, it is predicated with the U2 modal operator and therefore is inherited by the modal operator for advanced agents. In

```
*A (postcondition(A,configuration(pressure1,state(normal))) ->
        U3 ?S(subactions(A,S), nextAction(S,nil),
            postcondition(S,configuration(gauge1,state(within(range1))))
        )).
*A (postcondition(A,configuration(gauge1,state(within(range1)))) ->
        U2 motion(A, dial1, rotational)).
*A *O (type(0,pressure), postcondition(A,configuration(0,state(normal)))
        -> P2 purpose(A)).
*A *O (type(0,pressure), postcondition(A,configuration(0,state(normal)))
        -> P1 complex(A)).
```





Figure 5.37: Output: Step 2 for Advanced

terms of how agents view the action of normalizing pressure, beginner agents view it as being an action which serves as the purpose for doing other actions (P2 purpose(A)) and novice agents view it as being a complex action (P1 complex(A)).

The generation instance for advanced agents for Step 2 is the same as for Step 1, except  $t_1$  is replaced by  $t_2$  and a1 by a2. The generation process is also similar, as evidenced by the SPUD output window shown in Figure 5.37. (The lexical entry for the verb *normalize* is shown in Figure 5.38.)

The generation instance for beginner agents is shown in Figure 5.39. It is similar to the one for Step 1, except for the communicative goals. In this case, the goals of **concrete** and

Figure 5.38: Lexical entry for *normalize* 

Figure 5.39: Generation instance: Step 2 for Beginner

termination are added for the sub-action na1 to ensure that it is described appropriately. The unique feature of the generation process is that the beginner agents' view of the main action as a purposive action guides SPUD to use the ds3 discourse segment tree (see Figure 5.11) to put the sub-action in the main clause of the sentence. Then the non-fronted purpose clause (see Figure 5.12) must be used to express the purpose relation and the action information for the main action. SPUD's output, shown in Figure 5.40, shows the result of the generation process.

The generation instance for novice agents is the same as for beginner agents, except for the use of the P1 modal operator instead of P2. As in the generation of the Step 1 instructions for the novice agents, the fronted purpose clause construction is used to reflect



Figure 5.40: Output: Step 2 for Beginner

word = {	name =	<pre>{ clockwise }</pre>	
	decl = site =	<pre>{ beta(S,R,E) } { vp(S,R,E) }</pre>	
	 semantics =	<pre>{ pathDir(E, cw) }</pre>	
}.	trees =	<pre>{ bVPadv(S,R,E) }</pre>	
Tree er	ntry:	$vp(S,R,E)$ $vp(S,R,E)_* \diamond$	

Figure 5.41: Lexical entry for *clockwise* 

the novices' view of the main action as complex. The result of the generation process is shown in Figure 5.42. (The lexical entry for the adverb *clockwise* is shown in Figure 5.41.)

Even though Step 2 is a simpler action than Step 1, the generated instructions again vary in terms of the amount of action information presented to different agents as well as the style of presentation:

- (25) (a) Normalize pressure.
  - (b) Turn dial to normalize pressure.
  - (n) To normalize pressure, turn dial clockwise until gauge is within normal range.

One main point that the generation of Step 2 demonstrates is that SPUD's decision as to what will be the head of a construction (i.e. a high-level action versus a sub-action) depends on agent-specific features. Beginner agents' **purpose** view of the top-level action produces a sentence with a standard non-fronted purpose clause. Otherwise, the generation processes for the two steps are the same.

### 5.4.4 Starting the pump

The final step in restarting the pump, namely that of actually starting the pump, is similar to Step 1 in terms of its action/sub-action structure (see Figure 5.18). The action information is shown in Figure 5.43. The first sub-action, resetting the pump, has two subactions of its own: raising and holding the lever (see Figure 5.44). The second sub-action in the step, like its counterpart in Step 1, is a leaf in the action tree. However, unlike in Step 1, it has an explicit purpose relation to the main action. This difference, along with the differences in agent knowledge, results in instructions which differ in form from those in Step 1. The resulting instructions for Step 3 are:

- (26) (a) Start pump.
  - (b) To start pump, reset pump and then press ON button.
  - (n) Reset pump by moving lever to RESET position and then holding lever until the light is green. Then press ON button to start pump.



Figure 5.42: Output: Step 2 for Novice

```
% Action: a3
S step(a3).
S during (a3, t_3).
C result(a3, t_4).
S finishes(t_3, t).
S agent(a3, u).
S postcondition(a3, configuration(pump1, state(running))).
S subactions(a3, sa1).
S nextAction(a3, nil).
S prevAction(a3, a2).
S parent(a3, a).
% Action: sa1
S substep(sa1,a3).
S during(sa1, t_3_1).
S starts(t_3_1, t_3).
C result(sa1, t_3_2).
S agent(sa1, u).
S postcondition(sa1, configuration(pump1, state(reset))).
S subactions(sa1, ra1).
S nextAction(sa1, sa2).
S parent(sa1, a3).
% Action: sa2
S during(sa2, t_3_2).
S meets(t_{3_2}, t_{3_1}).
S finishes(t_3_2, t_3).
C result(sa2, t_4).
S agent(sa2, u).
S force(sa2, button1).
S magnitude(sa2, greater(resistance(button1))).
S pathDir(sa2, inDir(button1)).
S purpose(sa2, generate(a3)).
S nextAction(sa2, nil).
S prevAction(sa2, sa1).
S parent(sa2, a3).
```

Figure 5.43: Action information for starting the pump

```
% Action: ra1
S during(ra1, t_3_1_1).
S  starts(t_3_1_1, t_3_1).
C result(ra1, t_3_1_2).
S agent(ra1, u).
S motion(ra1, lever1, translational).
S pathEnd(ra1, at(resetPos1)).
S nextAction(ra1, ra2).
S prevAction(ra1, nil).
S parent(ra1, sa1).
% Action: ra2
S during(ra2, t_3_1_2).
S meets(t_3_1_2, t_3_1_1).
S finishes(t_3_1_2, t_3_1).
C result(ra2, t_3_2).
S agent(ra2, u).
S force(ra2, lever1).
S magnitude(ra2, weight(lever1)).
S pathDir(ra2, oppositeDir(gravity)).
S postcondition(ra2, configuration(light1, color(green))).
S nextAction(ra2, nil).
S prevAction(ra2, ra1).
S parent(ra2, sa1).
```

Figure 5.44: Action information for resetting the pump

```
*A (postcondition(A, configuration(pump1, state(running))) ->
     U3 ?S?N(subactions(A,S),
              postcondition(S,configuration(pump1,state(reset))),
              nextAction(S,N), nextAction(N,nil),
              force(N, button1), magnitude(N, greater(resistance(button1))),
              pathDir(N,inDir(button1)))).
*A (postcondition(A, configuration(pump1, state(reset))) ->
    U2 ?S?N(subactions(A,S),
             motion(S,lever1,translational), pathEnd(S,at(resetPos1)),
             nextAction(S,N), force(N,lever1), pathEnd(N,at(resetPos1)),
             pathDir(N,oppositeDir(gravity)), magnitude(N,weight(lever1)),
             postcondition(N,configuration(light1,color(green))))).
*A (postcondition(A, configuration(pump1, state(running))) ->
     P2 complex(A)).
*A (postcondition(A, configuration(pump1, state(running))) ->
      P1 elaborate(A)).
```

Figure 5.45: Agent knowledge for Step 3

The agent knowledge for Step 3 is shown in Figure 5.45. Advanced agents know that starting the pump involves resetting the pump and then pressing the ON button. Beginner agents know that resetting the pump involves raising the control lever to the RESET position and holding it until the status light turns green. As in the previous step, novice agents do not have any experience with actions like the top-level action. However, novice agents can be assumed to view such actions as **elaborate**. Beginner agents also have a particular view of such actions, in this case as **complex** actions. Both of these views are due to the fact that the main action has two full-fledged sub-actions (i.e. they not inferable from the top-level action by beginner and novice agents).

Many new verbs are needed for the generation of this step's instructions. These are shown in Figure 5.46.

The generation instance and generation process are as expected for the advanced agent type; the result is shown in Figure 5.47. The generation instances for beginners and

```
word = {
                           { start }
           name =
           . . .
                           \{ vp(S,R,E) \}
           site =
           . . .
                           { during(E,R),
           semantics =
                          postcondition(E,configuration(0,state(running))) }
           . . .
           pragmatics =
                           { type(0,pump) }
                           { transitiveVP(S,R,E,0) }
           trees =
                           { move }
word =
           name
           . . .
                           \{ vp(S,R,E) \}
           site =
           . . .
                           { during(E,R), motion(E,O,translational) }
           semantics =
           . . .
                           { property(0, movable, true) }
           pragmatics =
                           { transitiveVP(S,R,E,0) }
           trees =
                           { hold }
word = {
           name =
           . . .
                           \{ vp(S,R,E) \}
           site =
           . . .
                           { during(E,R), force(E,O),
           semantics =
                          magnitude(E,weight(0)),
                          pathDir(E,oppositeDir(gravity)) }
           . . .
                           { transitiveVP(S,R,E,0) }
           trees =
                           { press }
word =
           name =
           . . .
                           \{ vp(S,R,E) \}
           site =
           . . .
                           { during(E,R), force(E,0),
           semantics =
                          magnitude(E,greater(resistance(0))),
                          pathDir(E,inDir(0)) }
           . . .
                           { transitiveVP(S,R,E,0) }
           trees =
```

Figure 5.46: Verb entries for Step 3

Figure 5.47: Output: Step 3 for Advanced

novices are shown in Figure 5.48. The communicative goals again reflect the detail needed in the instructions for the less experienced agents.<sup>20</sup> In terms of generation, since beginner agents view the main action as complex, the fronted purpose clause construction is chosen to emphasize the main action and allow the sub-actions to be expressed in the main clause. The *and then* lexical item for conjoining verb phrases is chosen to add the action information for the second sub-action. The result of the generation for beginner agents is shown in Figure 5.49.

Finally, novice agents' view of the main action as elaborate guides SPUD to choose the discourse segment tree with two substitution sites for sentences, the second one introduced by "*Then*" (see Figure 5.16). Each sub-action is described in its own sentence, allowing for shorter sentences because each sub-action requires detailed description. The first sub-action is described by using a means clause to express how to accomplish it and *and then* to conjoin its two sub-actions. A purpose clause expressing the main action is adjoined to the verb phrase for the second sub-action, thus providing the necessary termination information to satisfy the termination(sa2) communicative goal. The result of the generation is shown in Figure 5.50.

<sup>&</sup>lt;sup>20</sup>Due to peculiarities of interacting goals, the purpose goal in the generation instance for novices was needed instead of the concrete and termination goals for sa2.

```
gen = {
  name = { Step 3 for Beginner }
  private = { S $ }
  shared = { P2 (P2 present(t_3) -> $) }
  describe = { ds(s,t_3,a3) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { concrete(a3) termination(a3)
                  concrete(sa1) termination(sa1) nextAction(sa1,sa2) }
}.
gen = \{
  name = { Step 3 for Novice }
  private = { S $ }
  shared = { P1 (P1 present(t_3) -> $) }
  describe = { ds(s,t_3,a3) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { subactions(sa1,ra1) nextAction(sa1,sa2)
                  concrete(ra1) termination(ra1) nextAction(ra1,ra2)
                  concrete(ra2) termination(ra2)
                  purpose(sa2,generate(a3)) }
```

```
}.
```

Figure 5.48: Generation instances: Step 3 for Beginner and Novice



Figure 5.49: Output: Step 3 for Beginner

### 5.5 Discussion

The agents' views and experiences are reflected in the variety of form and detail in the generated instructions. SPUD makes such a variety attainable with little extra effort beyond that required for generating only one instruction per action instance. Nearly all of the generated instructions described in this chapter have been produced by SPUD with the report "derivation completed successfully" (see Figure 5.17). The less-than-successful report by SPUD for instructions which seem to be fine is due to peculiarities in proving some communicative goals included in generation instances. Since the appropriate instructions are generated, working out why they do not receive SPUD's stamp of approval has not been a top priority. As could be expected from the variety in complexity of the generated texts, computational times for the generation of the texts differ considerably. The instructions for advanced agents, which are simple, generally take a couple of minutes, while the complex instructions for novice agents can take up to a half hour to generate.<sup>21</sup>

<sup>&</sup>lt;sup>21</sup>SPUD was run on a multi-user Sun server. Computation times include non-user (i.e. program) time for system tasks. No detailed information about computation times was collected as this is not a focus of

TFn	at pump by moutry leven to RESET pozition and then holding leven until light is ⊇neen. Than press OH puttion to start pump, of pump by muving leven to RESET musition and then holding leven until Light is green. Than press OH purnon to start pump.
-----	---

Figure 5.50: Output: Step 3 for Novice

Overall, the implementation in SPUD has been very successful.

SPUD's generation of the example instructions has demonstrated its generation algorithm as well as how the domain, agent, and action information all come together to enable the generation of tailored, effective instructions. The implementation of the language model for SPUD has involved specifying lexical items and their syntax, semantics, and pragmatics. The lexical items range from individual words to discourse segments, giving SPUD the flexibility to determine the structure of the generated text at all levels. By including agents' view of actions as part of the agent model, the pragmatics for lexical items can take account of the agent's view in deciding how to present a complex action. SPUD uses this pragmatic information when choosing lexical items to include in a description. In this way, SPUD tailors the descriptions to the agent. Communicative goals to be satisfied by the generated text are included in the generation instances which start the generation process. In checking if the communicative goals are satisfied, SPUD reasons about the agent's assumed knowledge and what information the current description provides to the agent. Thus, by specifying communicative goals relating to the concreteness and termination of the actions being described, the instructions generated by SPUD are effective and efficient.

In the course of implementing the language model, several issues arose which have been incorporated into this chapter. First of all, working with a action/sub-action structure from the top down means that syntactic trees have to be made such that the top-level action can be moved to a subordinate clause if needed (e.g. to emphasize a sub-action by putting it in the main clause). These are not the types of trees that are naturally intuitive to design but they are needed in addition to traditional trees and lexical entries which focus on the action described in the main clause as the main action to be described. Thus, the two purpose clause trees (Figures 5.12 and 5.13) move the top-level action from the main clause into a subordinate purpose clause.

Second, the implementation of stylistic preferences came from the need to "get around" SPUD's greedy algorithm which picks the simplest lexical entry, given the same semantics. Since instructions need to be described differently depending on agent expertise, these the dissertation. preferences (and the need to have them be private to each agent type) precipitated the need for additional modal operators to serve as private agent knowledge, i.e. knowledge that is not inherited through the experience level modal operators (U1, etc.). Using the stylistic preferences, implemented in the form of agents' views of actions, complex lexical items which would not otherwise be chosen by SPUD are now used.

Lastly, the communicative goals included in each action instance had to be different, reflecting the different levels of detail needed by different agent types. The problem with specifying a uniform set of communicative goals is that SPUD does not have a notion of partially satisfying a goal or of subgoaling. That is, each word has to completely satisfy a communicative goal in order to be considered [Stone, personal communication]. Thus, a rule such as

\*A C (concrete(A), termination(A) -> howToDo(A)).

and the communicative goal of howToDo(A) cannot be used. Instead, the two separate conjuncts of the howToDo(A) rule, namely concrete(A) and termination(A) must be used as communicative goals. In addition, rules cannot be used to go down more than one level in the action/sub-action tree. For instance, rules such as

```
*A *S C (subactions(A,S), concreteAll(S) -> concrete(A)).
```

C concreteAll(nil).

```
*A *N C (concrete(A), nextAction(A,N), concreteAll(N) -> concreteAll(A)).
```

are not effective in having the communicative goal concrete(a1) ensure that the resulting description will have the necessary concreteness information for a1's sub-actions. Therefore, separate communicative goals for each sub-action must be stated in generation instances.

Modifications to SPUD's algorithm which might make implementing a language model easier include changing its ranking method of possible additions to a tree and/or allowing multiple possibilities to be left open. Changing the ranking method could be used to give more weight to trees which provide more opportunities to include useful information later on. Leaving multiple options open when choosing which lexical item to add next could give the necessary flexibility when handling constructions with similar semantics. These and other possible modifications to SPUD are described as part of the conclusions and further work in Chapter 7. However, before concluding this dissertation, I present some related work in the area of Natural Language Generation and compare it to this implementation using SPUD.

# Chapter 6

# **Related Work**

Since my dissertation work involves a system for the generation of natural language instructions, comparing it to other such systems is the best way to distinguish its strengths and weaknesses. Thus, comparisons of my approach using SPUD and that of other systems used for generating instructions given at the beginning of this chapter. However, other generation approaches which do not directly address the generation of instructions are also discussed later. Even though direct comparisons cannot be made, either because the approach does not address instructions or because it is not a full system (or both), comparisons with these approaches can also show the strengths and weaknesses of my approach.

## 6.1 Instruction Generation Systems

Few of the instruction generation systems discussed below address the effectiveness of instructions; none of them address termination information. Few of the generation approaches below address the use of hearer knowledge, especially the level of expertise of the hearer, and few include the ability to reason about the world and the effects of instructions on the hearer. However, many address the importance of having a language-neutral domain/action representation and some incorporate incrementality.<sup>1</sup> All of these issues

<sup>&</sup>lt;sup>1</sup>Although not addressed in detail before, incrementality reflects a system's ability to consider and adapt to the information provided by partial descriptions.

— effectiveness, termination, hearer knowledge, reasoning, language-neutral domain representation, and incrementality — are the main points of discussion below. As such, each system is given a rating to indicate the degree to which they address each of these issues. A + that means the issue is addressed, a ~ means that the issue is addressed indirectly, and a - means that the issue is not addressed at all in the considered literature for the system. (A ? is used when no determination either way could be made.) The order of the issues is: effectiveness (termination), hearer knowledge (expertise levels), reasoning ability, language-neutrality, and incrementality. The issues in parentheses will also be given a rating when their associated issue has a + or ~ rating. My approach to instruction generation using SPUD would thus get a [+(+), +(+), +, +, +] rating.

# 6.1.1 COMET, 1990; rating:[-,-,~,+,+]

COMET [McKeown *et al.*, 1990] uses Functional Unification Formalism (FUF), a declarative and uniform representation, for domain and lexicon representation and unification for lexical choice and generation of multi-media explanations. Unification incrementally enriches the logical form determined by content planning until all aspects of the utterance are considered, giving COMET the ability to produce efficient texts. COMET can apply some reasoning, in the form of calling arbitrary code, in its generation process. However, it does not consider hearer knowledge nor the effectiveness of the generated text. (See [McKeown *et al.*, 1993], described in 6.2, for an extension of COMET's capabilities.)

# 6.1.2 EPICURE, 1992; rating:[~,~,-,+,+]

EPICURE [Dale, 1992] generates recipe instructions from a declarative feature-structure representation. Although the work focuses mainly on generating referring expressions, including determining when particular anaphoric forms (pronouns, reduced noun phrases, etc.) are appropriate, the approach to generation is part of the inspiration for the SPUD generator. A series of mapping algorithms transform semantic content into surface structure, making EPICURE incremental. It considers the ability of the hearer to understand which object a referring expression refers to in the domain, but it has a simple view of actions, reducing the complexity of actions found in the real world to state-change semantics. It does not address the ability to reason about hearer knowledge, the world, nor the effectiveness of the instructions beyond being able to identify the intended objects.

# 6.1.3 TECHDOC, 1994; rating:[~,~,+,+,-]

TECHDOC [Rösner and Stede, 1994] generates descriptions and instructions needed for maintenance activities. The description logic LOOM, supports some reasoning, is the representation for text structuring information as well as domain knowledge, which is language-neutral. *Penman*, a systemic-functional sentence-level generator, is used for sentence planning and lexical choice. Penman consists of system networks which encode various semantic and syntactic features and associated realization statements which indicate that particular words or constructions are to be used. Penman does not seem to be incremental in its generation, as a pass through the networks indicates the surface form features which must be then reconciled with the grammar. Since TECHDOC simulates the events it is to describe and updates the world model accordingly, the system is sensitive to the state of the world, i.e. only relevant information is provided in instructions. However, it does not address hearer knowledge nor the effectiveness of the instructions, beyond being appropriate to the state of the world.

## 6.1.4 IDAS, 1995; *rating:*[~,+(+),~,-,?]

IDAS [Reiter *et al.*, 1995] generates on-line technical documentation from domain knowledge bases developed for design purposes (e.g. computer-aided design output). The action representation for "deep" generation is case frames (roughly, predicate-argument structures), which by their very nature have a language bias. However, in order to lower costs in terms of development time, it uses a hybrid action representation in the form of canned text with embedded knowledge-base references and case frames with textual case fillers. In this way, domain developers, who are not experts in knowledge representation, can easily specify the language that goes along with the actions in the domain. A description logic representation is used for all information, including the grammar and lexicon. Reasoning thus is possible but limited because the description logic is a KL-ONE type
object classification system, which only supports reasoning about objects and their place in the classification hierarchy. SPL (sentence planning language) expressions are built from content-determination output which is sensitive to user expertise.

Lexical choice follows [Reiter, 1991], which is an algorithm for choosing appropriate noun phrases, and the generation is sensitive to the user model, which is provided as part of the input. In [Reiter, 1991], nouns are chosen by searching for lexical units that are known to the user, that truthfully describe the object, that convey sufficient information to satisfy communicative goals, and that are maximal under a lexical preference function. [Reiter, 1991] separates what the system knows from what it wishes to communicate to the hearer. The algorithm relies on representing human mental categories, since the lexical knowledge for individual users includes the lexicon and mental categories. (A one-to-one mapping between concepts and words is not necessarily assumed to exist in the mind of the hearer.) Communicative goals, such as identifying an object, are analyzed by a separate component and given to the lexical choice algorithm as attributes to express. Lexical preference is based on a bias towards basic-level and other preferred lexical units following [Rosch, 1978].

One of the basic questions that can be asked of the system is how to perform a specific action; other questions involve describing objects and relationships between objects in the domain. Other than considering user expertise in content-determination and lexical choice, the issue of the effectiveness of the instructions provided in response to the "how to perform an action" questions is not addressed.

### 6.1.5 IMAGENE, 1995; *rating:*[~(~),-,~,-,?]

[Vander Linden and Martin, 1995] uses a system network and sentence-building component on top of Penman (see Section 6.1.3) to generate technical documentation. The focus of the generation is on the realization of purpose expressions in instructions [Vander Linden, 1994]. Lexical choice (in this case, determining the grammatical form of purpose relations) is done by system networks. The system networks, which encode decisions derived from a corpus analysis of instruction manuals, make choices ranging in scope from discourse to sentences and phrases. The action representation is in the description logic LOOM and therefore some reasoning can be done by the system. However, the action representation includes some lexical information and therefore is not language-neutral. Contextual factors include interpersonal as well as discourse factors; however, hearer knowledge is not considered. Since the system is devoted to the realization of appropriate expressions of purpose, the effectiveness of the generated instructions is addressed indirectly. As shown in my approach using SPUD, purpose expressions are often used to convey information, including termination, needed for performing actions.

## 6.1.6 GhostWriter, 1996; rating:[-,-,-,~,?]

GhostWriter [Merchant *et al.*, 1996] uses a knowledge-based model of plans and actions in language-neutral form as basis for the semi-automatic generation of instructions. An explicit fine-grained action representation is used, making it mostly language-independent. However, actions can have a linguistically-oriented representation associated with them. In addition, there are concept-lexeme mapping structures in the lexicon. Action schemas are used for building a plan, which then can be used as input to the generator. The author can manipulate the plan, have the system generate instructions for the plan, and then post-edit the instructions by having the system re-generate parts of the instructions. Hearer knowledge is not addressed, nor is the effectiveness of the generated instructions, beyond the author's intervention by post-editing.

# 6.2 Other approaches in Natural Language Generation

Below I describe other approaches in NLG which are either methods for generating instructions that are not part of complete systems or methods for generating other types of texts which are related to instructions or which employ a user model. I only note what the approaches do since these are not directly comparable to my implementation.

#### 6.2.1 Paris, 1988

[Paris, 1988] describes TAILOR, a question-answering system in the domain of complex patented devices. The goal of describing a device is to help the user construct a mental functional model of it. The focus is on the content of the description, not its phrasing. Two ways of describing objects were found: one centered around subparts of the objects and the other around the processes which the objects can undergo. Which description strategy to use depends on the user's domain knowledge, since domain knowledge about specific objects or basic underlying concepts can affect an object's description. Users may have global expertise, in the form of known basic concepts, as well as local expertise, in the form of known objects. The description strategies can be mixed, based on the user's knowledge, by switching from one to the other at decision points in the Augmented Transition Networks which determine the content to be realized in the generated text. In this way, the kind of information, and not just the amount, in a generated text is affected by the user's level of expertise in the domain.

#### 6.2.2 Mellish and Evans, 1989

[Mellish and Evans, 1989] present a system which generates text from plans produced by the NONLIN planner, a non-linear planner, in the domains of cooking and maintenance activities. The plans which form the input to the generator are formally defined and domain-independent; they are non-linear action graphs and include the history of the hierarchical expansion of the nodes during planning. The generation process uses simple, well-understood, and restricted computational techniques such as recursive descent traversal. The plans are converted into "messages" in the content-planning phase; messages are a linguistically-motivated intermediate representation from which linguistic structures are built using rules. The resulting texts explain the actions to be performed and why they have to be done that way. [Mellish and Evans, 1989] states that the texts lack the smoothness of natural text due to a number of factors, including that the plans have a rich structure but not the kind needed for interesting Natural Language, that the plans do not reflect "human" organization of actions, that the use of a restricted view of the world which does not match up with natural semantics, and that the range of expressions is restricted by the planner's representation.

#### 6.2.3 Bateman and Paris, 1989

[Bateman and Paris, 1989] describe an approach designed to ensure that the generated text is appropriate for the intended audience. Since misphrased texts can be as ineffective as texts which wrongly direct attention or rely on non-existent hearer knowledge, they point out that beyond content determination, phrasing must be tailored to the specific user type. They introduce a second phase of planning to take the given content and determine how to control, as oppose to achieve, the appropriate phrasing. *Registers*, or specifications of the linguistic consequences of using language in particular situations, encode the fact that syntactic patterns and lexical features can be different for different user groups. They construct sets of terms to use for each user group, where the user groups are system developers, end users, and naive users of an expert system. Each group differs in its goals, expectations, and expertise with the expert system and the domain. The registers control how Sentence Planning Language (SPL) expressions are created and which grammatical features are allowable; in turn, the SPL expressions control how the generator, Nigel, produces text.

#### 6.2.4 Scott and de Souza, 1990

[Scott and de Souza, 1990] rely on Rhetorical Structure Theory (RST) to structure sentences and overall text. (RST is a method for describing relationships, i.e. *rhetorical relations*, between spans of text.) They promote the use of accurate and unambiguous markers (e.g. cue words) of rhetorical relations to make sure the intended message gets across to the hearer despite the lack of a good hearer model. They use heuristics to implement lexical choice with respect to choosing the most appropriate rhetorical relation to lexicalize for the given semantic content. Thus, they indirectly address the effectiveness of the generated text, but they do not address hearer knowledge since they assume the lack of a good hearer model.

#### 6.2.5 Rösner and Stede, 1992

As part of the TECHDOC project (see 6.1.3), [Rösner and Stede, 1992] approached the problem of analyzing naturally occurring instructions (from car manuals) using Rhetorical

Structure Theory (RST). They point out several problems with using traditional RST analyses on instructions. First, the RST idea of a "minimal unit" is unclear when it comes to instructions; that is, is it a clause, a sentence, an entire instruction step? Second, more than one rhetorical relation can often apply between two text spans in an instruction; traditional RST disallows this possibility. Finally, RST is incapable of representing the complex dependencies that appear in instructions. In attempting to overcome some of these problems, [Rösner and Stede, 1992] propose new RST relations for the genre of instructions: precondition, where the satellite proposition must be realized for the nucleus action to be successful when carried out; *until*, where the nucleus action is carried out as long as the satellite, the "stopping condition", is not yet true (the stopping condition may result from the nucleus action); and *step-sequence*, which is for the sequence of instruction steps which make up a task (as opposed to a normal *sequence* relation which is for a sequence of actions making up an instruction step). They encode domain knowledge and macrostructure "schematas" in LOOM and attach methods to schemata objects for building RST trees. which should be language-independent. They identify principles which make realizations of the RST relations more acceptable and use them to choose the possible RST structure with the highest preference after generating all such possible structures.

#### 6.2.6 McKeown, Robin, and Tanenblatt, 1993

[McKeown *et al.*, 1993] describes strategies in COMET (see above) to avoid using words that are not known by the user. Since COMET is a multi-media explanation system, unknown words are frequently disambiguated by accompanying illustrations. However, when an illustration is not sufficient, COMET uses four strategies to avoid using unknown words: selecting alternative words or phrases, rephrasing using conceptual definitions, rephrasing using descriptive referring expressions, or using past discourse to construct a referring expression. Lexical choice is part of the text generator and it depends on the user model, past discourse, syntactic form, and lexical constraints. The user model includes the user's reading level, unknown words, and wording preferences. Lexicon entries associate semantic concepts with words used to realize them and the lexical choice algorithm determines grammatical form based on semantic structure. If none of the strategies work to disambiguate an unknown word, the content planner must be reinvoked since different levels of knowledge (i.e. coarse- vs. fine-grained) are represented separately in the knowledge base.

#### 6.2.7 Kosseim and Lapalme, 1995

[Kosseim and Lapalme, 1995] address a restricted form of lexical choice, that of choosing which rhetorical relations to use when mapping a semantic representation to a rhetorical structure. Thus, they focus on the choice of linguistic constructions (e.g. those expressing rhetorical relations, such as means or purpose) rather than on the choice of individual words (except those associated with the linguistic constructions). They use heuristics, derived from a corpus analysis, to determine the realization of two *semantic carriers*, effects and guidances, as rhetorical relations. In that they address effects and guidances in instructions, they indirectly address the effectiveness of the instructions.

#### 6.2.8 Nicolov, Mellish, and Ritchie, 1996

[Nicolov *et al.*, 1996] exploit the declarative relationship between a non-hierarchical semantic representation, in the form of conceptual graphs, and a linguistically-motivated syntactic representation. Conceptual graphs are a language-neutral domain representation. D-Trees, a variation of TAG, are used for the lexicon and grammar. Their approach to generation involves incrementally finding mapping rules (semantics to syntax) to cover as much of the semantics in a conceptual graph as possible while adding as little *extra* information to the resulting text as possible. Their method allows the linguistic realization of a conceptual graph to be spread over multiple sentences. When the semantics of paraphrases are the same, they employ syntactic/stylistic preferences to choose which paraphrase to generate. In many ways, this approach is similar to that of SPUD, including incrementality and the use of a TAG-based grammar. However, even though a languageneutral domain representation is used, it is unclear whether conceptual graphs could fully support the specification of actions like those used in the example domain implementation in this dissertation.

# 6.3 Discussion

Why use SPUD as opposed to one of the other systems discussed in this chapter? As shown by the ratings of the other systems, all of them fail to fully address certain issues that I have argued as being essential to generation of effective instructions. Beyond my specific contributions with respect to the action representation and what action information is needed for effective instructions, what distinguishes my approach is SPUD's strengths as a generation system.

SPUD has a powerful reasoning system, based on an independently-motivated representation (modal First-Order Logic). Not only does this support language-independent reasoning about the domain, e.g. for planning or simulation purposes, it uses reasoning to determine the information (and its consequences) conveyed to the hearer by partial descriptions. Such reasoning, combined with the close coupling of syntax and semantics in its language representation, produces effective and efficient texts, i.e. those that convey the necessary information and only the necessary information. Since SPUD does sentence planning and syntactic realization at the same time, the overall generation process is flexible and efficient, responding to changes in domain and hearer knowledge with ease.

Despite its strengths, however, I encountered a few problems with SPUD in the implementation of the example domain (see Section 5.5). Therefore, in the concluding chapter, I suggest how SPUD could be modified in order to expedite implementation and generation of instructions, as well as other types of texts.

# Chapter 7

# **Conclusions and Further Work**

In this dissertation, I have supported my claim that the generation of effective instructions relies on representing complete action information, ensuring that all necessary information about an action is available from its description, and taking into account what the hearer is assumed to know. I have presented a corpus study of maintenance instructions which shows how one particularly vital piece of information, action termination, is expressed. I have motivated an action representation based on insight gained about actions from the corpus study as well as the representational needs of simulating agents carrying out similar tasks. I have shown how knowledge about domains, actions, and agent expertise in the domain can be represented and reasoned about by a modal first-order logic theorem-prover. Finally, I have brought all of this work together by using the Natural Language generator SPUD to generate effective instructions for different agent expertise levels.

In each of the areas that this dissertation addresses, I have made contributions to the current research while relying on previous work of other researchers. In this chapter, I summarize these contributions and raise some issues for further work. Overall, I believe that this dissertation demonstrates the potential for a Natural Language Generation system which produces effective instructions using an independently-motivated representation for domains, actions, and agents as well as a linguistically-sound lexical representation.

The results of the corpus study discussed in Chapter 3 shows that termination information is an important part of maintenance instructions. A third of the instructions in the corpus have termination information supplied in some form other than the verb; the actions described in these instructions do not naturally have termination. The simple ways of expressing termination, such as verb arguments and prepositional phrases, dominate the complicated ways (with the exception of purpose clauses) which add additional clauses. Since purpose clauses can convey a variety of information, including purpose, manner, and termination, using a purpose clause to convey multiple pieces of information is efficient. Thus purpose clauses are used frequently even though they are syntactically complex. Expressing termination using verb arguments is syntactically simple, e.g. *raise the lever*, but determining that particular verbs with particular arguments have termination information is hard.<sup>1</sup> Thus, in the interest of being able to implement the results of the corpus analysis, the syntactically complicated but semantically simple ways of expressing termination are analyzed in more detail for inclusion in the implemented language model.

In particular, the analysis demonstrates how and why prepositional phrases, purpose clauses, and *until* clauses are used to express termination, as summarized below.

- **Prepositional phrases:** The prevalence of actions involving motion in the corpus of maintenance instructions means that prepositional phrases are frequently needed to express path information, particularly path endpoints for the termination of the motion. Such prepositional phrases are semantically simple, simply adding the motion's endpoint to the action information; in addition, they are syntactically simple, requiring only an additional phrase (as opposed to a clause).
- **Purpose clauses:** As mentioned before, purpose clauses can provide a variety of action information. The corpus analysis includes *means* ("by") clauses with purpose clauses, since means clauses are express the same action/sub-action type of relation but in a different syntactic order. Means clauses are used frequently since they express the action(s) used to accomplish another action.

Although not noted explicitly in the corpus coding, two forms of the "to" purpose clause, fronted and non-fronted, appear in the corpus. Fronted purpose clauses, at the beginning of a sentence, provide a framework for describing and interpreting the actions in the sentence's main clause which accomplish the purpose. Non-fronted

<sup>&</sup>lt;sup>1</sup>At the very least, reasoning about the interaction of action properties and arbitrary properties of the domain would be required.

purpose clauses, on the other hand, simply express a local purpose role relating an action directly to its purpose. This type of purpose clause can be used when both the purpose and its accomplishing action are expressed simply.

Purpose clauses are syntactically complicated, adding another clause to a sentence. However, they provide so much action information that using them in a description reduces the effort in describing and interpreting the action description. Thus, purpose clauses are used frequently in the corpus of maintenance instructions and are worth the effort to include in the language model.

**Until clauses:** Finally, although *until* clauses do not appear frequently in the corpus, they are one of the simplest ways of adding termination information to an action description. By expressing a termination condition which indicates when to stop performing an action, *until* clauses provide explicit termination. However, since this is all they provide, with no other indication of how the termination information is related to the performance of the action, they require additional work on the part of the hearer to interpret. Therefore, this relatively simple way of providing termination information is infrequent in the corpus but is included in the language model because of its semantic simplicity.

Much more work needs to be done to understand all of the ways in which termination information is expressed. Verb arguments, free adjuncts, expressions of manner, as well as non-lexical sources of termination, all have to be explored in terms of their capacity to express termination. The corpus study presented in this dissertation is a step in the right direction, in addition to providing the basis for the implemented language model and insight into what kind of action information needs to be represented.

While the action information needed by Natural Language instructions can motivate an action representation, having an action representation which can be used easily in other applications, such as planning or simulation, ensures that it is well-motivated and independent of any theory of language. A Natural Language Generation system which uses such a representation is more likely to succeed since without a language-neutral action representation, a separate step would be needed to translate the source action information (e.g. from a planning system) into the NLG system's action representation. To avoid this additional step and to simplify the generation process, the action representation should supply the primitives for lexical representation as well as non-linguistic applications. By adopting an action representation which is flexible and general enough to support both types of representational needs, the need to develop and maintain disparate representations disappears.

In Chapter 4, I presented an action representation (PAR) meant to be language-neutral and universal, developed with researchers in both the simulation of virtual humans and the semantics of Natural Language. The requirements for its structure come from the need to represent all aspects of actions which are vital to their performance. This makes PAR an ideal representation for the generation of effective instructions, since all the necessary action information is represented. By translating PAR from its simple feature-structure implementation to a first-order logic implementation, rules can be used to formalize reasoning about actions. In the implementation, rules formalize the ways in which termination information is derived from an action's specification. In addition, rules about the concreteness, or performability, of actions formalize reasoning required to ensure that an action's specification is adequate for its performance. In this way, reasoning about action information has been formalized in a language-independent manner. While PAR and its implementation in first-order logic have proven suitable for the implementation presented in this dissertation, it would benefit from a more sophisticated action semantics, closer to the dynamic semantics and tripartite structure discussed in Section 2.1. Changing PAR to reflect this more formal treatment of actions would be a significant contribution in the area of action representation and reasoning and is a topic for further work.

Taking advantage of the representational and reasoning power of modal first-order logic, used by SPUD as its knowledge representation language, knowledge about agent expertise has been encoded and provided to SPUD as described in Chapter 4. Modal operators are used to predicate knowledge assumed to be known by certain types of agents and to formalize the inheritance of knowledge, reflecting the accumulation of agent expertise with experience. Three types of agents are implemented: novice, beginner, and advanced. Each has its own knowledge of the domain, in addition to inheriting knowledge from the agent type below it. In this way, agent models are built and given to SPUD to use in reasoning about the best way to describe actions to particular agents.

In addition to domain, action, and agent knowledge, a language model has been implemented for use by SPUD as described in Chapter 5. Using the same predicates as in the domain, action, and agent representations, linguistic constructions have been encoded for use in describing the maintenance activities considered in this dissertation. These include entries for discourse segment structures, sentence structures, verbs, prepositional phrases, purpose clauses, and *until* clauses. Some of these entries are not linguistically intuitive, since they are designed for descriptions which start with higher-level actions and need to be rearranged to emphasize lower-level ones. The pragmatics for these entries indicate when they are appropriate to use, so that SPUD can make intelligent decisions when choosing lexical items to add to a description.

However, in order for SPUD to make the right decisions when choosing between semantically similar constructions, more information must be provided. In this implementation, knowledge about how agents are assumed to view certain kinds of actions is provided to SPUD (Section 5.2.1). These views are implemented as knowledge predicated with private, non-inherited modal operators; such knowledge is then used in the pragmatics of syntactic constructions. With this guidance, SPUD chooses the most appropriate constructions despite the fact that they might otherwise be overlooked in the generation process. This type of guidance is essential given the way in which SPUD generates text. The implementation of the stylistic preferences came late in the research process and further work would refine and formalize their use.

This implementation has used SPUD as the Natural Language generator which produces effective instructions when given action information and rules about action termination and concreteness, the syntax and semantics of linguistics constructions derived from the corpus analysis, and the knowledge that agents are assumed to have. In many areas it exceeded my expectations in what it could do, but in some it fell short. In terms of evaluating SPUD's generation of instructions, I consider whether effective instructions, with the form and level of detail appropriate for the particular agent, are generated to be the primary criteria. Secondary criteria include SPUD's status report for the final generated text and the computational time required for the generation. Using these evaluation criteria, SPUD performed very well.

In terms of scalability of the implementation, what is most impressive is the relative ease with which complex syntax and semantics could be encoded and used. New lexical items and syntactic constructions could be added with little effort in order to implement new constructions. In this way, more types of instructions could be included in the implementation. Similarly impressive is the power of SPUD's theorem-prover, used for reasoning about the effects of using particular linguistic constructions on the hearer's knowledge. The richness of the domain representation enables the formalization of agent expertise levels as well as reasoning rules about action termination and concreteness, thus enabling the reasoning about an instruction's effectiveness. One issue that might affect scalability is computational time, since the more facts and rules that make up the domain model, the more time the theorem-prover could require to reason about the domain. Careful attention to the encoding of rules could partially alleviate this potential problem. The generality and power of SPUD's knowledge representation (modal First-Order Logic) lends itself well to increasing the complexity of the implemented domain or modeling a new domain. Such reasoning power gives SPUD the flexibility to generate different instructions based on agent expertise for the same action.

A problem arose in implementing the example domain and the corresponding instructions. Since SPUD considers the addition of only one lexical item at a time, any rules used to reason about the addition of lexical items can in practice only justify one lexical item at a time. A rule such as

## \*A C (concrete(A), termination(A) -> howToDo(A)).

cannot be used to ensure both the concreteness of an action's specification and the inclusion of termination information (as defined in Sections 4.3.2 and 4.3.3). Such a rule would require the addition of multiple lexical items' semantics to satisfy it. Since no single lexical item's semantics satisfies this rule, it could not be used to justify the addition of the appropriate *multiple* lexical items. Complex communicative goals, such as howToDo(A) as defined above, cannot be used in generation instances because of this property of SPUD. Multiple communicative goals are thus required in generation instances, unpacking the information so that SPUD can use it to generate the appropriate instructions.

Complex communicative goals, which simplify the specification of generation instances but need multiple lexical items to satisfy, suggest opportunities for improving or extending the way SPUD works. One possibility is to implement dynamic communicative goals, i.e. goals that can be posted on the fly when reasoning about other goals. In this way, the multiple communicative goals needed to provide the correct generation behavior would not need to be specified in advance. Another way to achieve this would be to add a contentplanning component to SPUD, since communicative goals form part of the specification of the content. SPUD was not designed to plan the content of the generated text [Matthew Stone, personal communication and so generation instances, used to tell SPUD what content to convey, are meant to come from another, hopefully automated, component of a Natural Language Generation system. Finally, a less drastic but more fundamental change would be to modify the search algorithm that SPUD employs. SPUD currently uses a greedy search algorithm which chooses the single best lexical item to add to the current tree. Algorithms such as beam search, which explore multiple paths (in this case, sequences of tree operations) simultaneously, could be used to give SPUD the ability to look ahead for combinations of lexical items that would satisfy a complex communicative goal. Any of these modifications could solve the problem of complex communicative goals but implementing them would require much further work.

In the implementation described in this dissertation, I circumvented the problem of complex communicative goals by using multiple communicative goals in the generation instances for the example instructions. Using this hand-constructed content, which could eventually come from an automated content-planner or dynamically-posted communicative goals, I have shown how SPUD can be used to generate effective instructions, sensitive to the inclusion of action termination and other information necessary for action performance. I have also shown how SPUD can take an agent model, indicating an agent's expertise with the domain, and tailor the generated instructions to the agent, ensuring that the instructions are *effective for that agent*. In conclusion, this dissertation has demonstrated that effective instructions can be generated when the right action information, agent information, and language information are brought together and reasoned about in a Natural Language generator which, most essentially, considers the effect of every word that it includes in a description.

# Bibliography

- [Allen, 1983] J. Allen. Maintaining knowledge about temporal intervals. Communications of the ACM, 26:832-843, 1983.
- [Allen, 1984] J. Allen. Towards a general theory of action and time. Artificial Intelligence, 23:123–154, 1984.
- [Badler et al., 1998] N. Badler, R. Bindiginavale, J. Bourne, M. Palmer, J. Shi, and W. Schuler. A parameterized action representation for virtual human agents. In Workshop on Embodied Conversational Characters (WECC98), 1998. Lake Tahoe, CA, October 12–15.
- [Badler et al., 1999] N. Badler, R. Bindiganavale, J. Bourne, J. Allbeck, J. Shi, and M. Palmer. Real-time virtual humans. In Proceedings of the International Conference on Digital Media Futures, Bradford, UK, 1999.
- [Badler et al, 1997] Badler et al. Natural Language Text Generation from Task Networks for Use in Automatic Generation of Technical Orders from DEPTH Simulations. Final Report to Air Force HRGA, March 1997.
- [Badler et al, 1998] Badler et al. Automating maintenance instructions. Final Report to Air Force HRGA, March 1998.
- [Bateman and Paris, 1989] J. Bateman and C. Paris. Phrasing a text in terms the user can understand. In *Proceedings of IJCAI-1989*, 1989.
- [Bleam et al., 1998] T. Bleam, M. Palmer, and K. Vijay-Shanker. Motion verbs and semantic features in TAG. In TAG+4 Workshop, 1998. Philadelphia, PA, August 1–3.

- [Dale, 1992] R. Dale. Generating Referring Expressions: Constructing Descriptions in a Domain of Objects and Processes. ACL-MIT Press Series in Natural Language Processing. MIT Press, 1992.
- [Dang et al., 1997] H. Dang, J. Rosenzweig, and M. Palmer. Associating semantic components with levin classes. In Proceedings of Interlinga Workshop, MTSUMMIT97, San Diego, CA, October 1997.
- [Dang et al., 1998] H. Dang, K. Kipper, M. Palmer, and J. Rosenzweig. Investigating regular sense extensions based on intersective levin classes. In Proceedings of Coling-ACL98, August 1998.
- [Di Eugenio and Webber, 1996] B. Di Eugenio and B. L. Webber. Pragmatic overloading in Natural Language instructions. International Journal of Expert Systems, Special Issue on Knowledge Representation and Reasoning for Natural Language Processing, 9(1), March 1996.
- [Di Eugenio, 1993] B. Di Eugenio. Understanding Natural Language Instructions: A Computational Approach to Purpose Clauses. PhD thesis, University of Pennsylvania, 1993.
- [Dixon, 1987] P. Dixon. Actions and procedural directions. In Coherence and Grounding in Discourse: outcome of a symposium, Eugene, Oregeon, June, 1984. J. Benjamins Pub. Co., Amsterdam; Phildelphia, 1987.
- [Elhadad et al., 1997] M. Elhadad, K. McKeown, and J. Robin. Floating constraints in lexical choice. Computational Linguistics, 23(2):195–239, 1997.
- [Grice, 1975] H. P. Grice. Logic and conversation. In P Cole and J L Morgan, editors, Syntax and Semantics, volume 3: Speech Acts, pages 1–58. Academic Press, 1975.
- [Hartley and Paris, 1996] A. Hartley and C. Paris. Two sources of control over the generation of software instructions. In *Proceedings of the 34th Annual Meeting*. Association for Computational Linguistics, June 1996.
- [How, 1993] K. Y. How. A Processing Framework for Temporal Analysis and Its Application to Instructional Texts. PhD thesis, University of Edinburgh, 1993.

- [ITL SIMA, 1997] ITL SIMA. Visual Interface to Manufacturing: Black&Decker/ITL Visualization SIMA Project. Web Site, 1997. http://speckle.ncsl.nist.gov/~sima/vim/.
- [Joshi et al., 1975] A. Joshi, L. Levy, and M. Takahashi. Tree Adjunct Grammars. Journal of Computer and System Sciences, 1975.
- [Kosseim and Lapalme, 1995] L. Kosseim and G. Lapalme. Choosing rhetorical relations in instructional texts: The case of effects and guidances. In Proceedings of the Fifth European Workshop on Natural Language Generation, May 1995.
- [McKeown et al., 1990] K. McKeown, M. Elhadad, Y. Fukumoto, J. Lim, C. Lombardi, J. Robin, and F. Smadja. Natural language generation in COMET. In Robert Dale, Chris Mellish, and Michael Zock, editors, *Current Research in Natural Language Generation*, Cognitive Science Series, chapter 5. Academic Press, Harcourt Brace Jovanovich, 1990.
- [McKeown et al., 1993] K. McKeown, J. Robin, and M. Tanenblatt. Tailoring lexical choice to the user's vocabulary in multimedia explanation. In Proceedings of ACL-93, 1993.
- [Mellish and Evans, 1989] C. Mellish and R. Evans. Natural language generation from plans. *Computational Linguistics*, 15(4):233-249, 1989.
- [Merchant et al., 1996] B. P. Merchant, F. Cerbah, and C. S. Mellish. The GhostWriter Project: a demonstration of the use of AI techniques in the production of technical publications. In A. Macintosh and C. Cooper, editors, Applications and Innovations in Expert Systems IV. SGES Publications, 1996.
- [Moens and Steedman, 1987] M. Moens and M. Steedman. Temporal ontology in natural language. In Proceedings of the 25th Annual Meeting of the ACL, pages 1–7, 1987.
- [Moens, 1987] M. Moens. Tense, Aspect and Temporal Reference. PhD thesis, University of Edinburgh, 1987.
- [Mourelatos, 1981] A. Mourelatos. Events, processes, and states. In P J Tedeschi and A Zaenen, editors, *Tense and Aspect*, volume 14 of *Syntax and Semantics*. Academic Press, 1981.

- [Nicolov et al., 1996] N. Nicolov, C. Mellish, and G. Ritchie. Approximate generation from non-hierarchical representations. In Proceedings of the Eighth International Workshop on Natural Language Generation, 1996.
- [Palmer et al., 1997] M. Palmer, J. Rosenzweig, H. Dang, and F. Xia. Capturing syntactic/semantic generalizations in a lexicalized grammar. presentation in working session of Semantic Tagging Workshop, ANLP-97, 1997.
- [Palmer et al., 1999] M. Palmer, J. Rosenzweig, and W. Schuler. Capturing motion verb generalizations with Synchronous TAGs. In Patrick St. Dizier, editor, *Predicative Forms* in NLP. Kluwer Press, 1999.
- [Paris, 1988] C. Paris. Tailoring object description to a user's level of expertise. *Computational Linguistics*, 14(3), 1988.
- [Passonneau, 1987] R. J. Passonneau. Situations and intervals. In Proceedings of the 25th Annual Meeting of the ACL, pages 16–24, 1987.
- [Reader's Digest, 1991] Reader's Digest. New Complete Do-It-Yourself Manual. Reader's Digest Association, 1991.
- [Reiter et al., 1995] E. Reiter, C. Mellish, and J. Levine. Automatic generation of technical documentation. Applied Artificial Intelligence, 9, 1995.
- [Reiter, 1991] E. Reiter. A new model of lexical choice for nouns. Computational Intelligence, 7(4):240–251, 1991.
- [Rosch, 1978] E. Rosch. Principles of categorization. In Eleanor Rosch and Barbara B. Lloyd, editors, *Cognition and Categorization*, pages 27–48. Erlbaum, Hillsdale, NJ, 1978.
- [Rösner and Stede, 1992] D. Rösner and M. Stede. Customizing RST for the automatic production of technical manuals. In Aspects of Automated Natural Language Generation (6th International Workshop on NLG), 1992.
- [Rösner and Stede, 1994] D. Rösner and M. Stede. Generating Multilingual Documents from a Knowledge Base: The TECHDOC Project. In Proceedings of COLING'94, 1994.

- [Schabes, 1990] Y. Schabes. Mathematical and Computational Aspects of Lexicalized Grammars. PhD thesis, Computer Science Department, University of Pennsylvania, 1990.
- [Scott and de Souza, 1990] D. R. Scott and C. S. de Souza. Getting the message across in RST-based text generation. In Robert Dale, Chris Mellish, and Michael Zock, editors, *Current Research in Natural Language Generation*, Cognitive Science Series, chapter 3. Academic Press, Harcourt Brace Jovanovich, 1990.
- [Steedman, 1997] M. Steedman. Temporality. In J van Benthem and A ter Meulen, editors, Handbook of Logic and Language. Elsevier North Holland, 1997.
- [Stone and Doran, 1997] M. Stone and C. Doran. Sentence planning as description using Tree-Adjoining Grammar. In Proceedings of ACL/EACL'97, pages 198–205, 1997.
- [Stone and Webber, 1998] M. Stone and B. Webber. Textual economy through close coupling of syntax and semantics. International Workshop of Natural Language Generation, August 1998.
- [Stone, 1997] M. Stone. Applying theories of communicative action in generation using logic programming. In Proceedings of the AAAI 1997 Fall Symposium on Communicative Action in Human and Machines, Cambridge, MA, November 1997.
- [Stone, 1998] M. Stone. Modality in Dialogue: Planning, Pragmatics and Computation. PhD thesis, University of Pennsylvania, 1998. Tech Report IRCS-98-23.
- [Thompson, 1985] S. A. Thompson. Grammar and written discourse: Initial vs. final purpose clauses in english. *Text*, 5(1-2), 1985.
- [USAF, 1988] USAF. Organizational Maintenance Job Guide (Fuel System Distribution, USAF Series F-16C/D Aircraft). Technical Order Manual, 1988.
- [Vander Linden and Martin, 1995] K. Vander Linden and J. H. Martin. Expressing rhetorical relations in instructional text: A case study of the purpose relation. *Computational Linguistics*, 21(1):29–57, 1995.

- [Vander Linden, 1994] K. Vander Linden. Generating precondition expressions in instructional text. In Proceedings of the 32nd Annual Meeting of the ACL, 1994.
- [Vendler, 1967] Z. Vendler. Verbs and times. In Linguistics in Philosophy, pages 91–121. Cornell University Press, 1967.

# Appendix A SPUD files

These are the full SPUD input files for this implementation.

# A.1 Lexical items

```
%% Discourse segment entries
% Normal
word = {
  name = {  \setminus .  }
 basic = { true }
  decl = { alpha(S,R,E) }
  site = { ds(S,R,E) }
  match = \{ () \}
  semantics = { step(E), during(E,R) }
  presupposition = { true }
  pragmatics = { true }
 trees = { ds1(S,R,E) }
}.
% Multi-sentence
word = {
  name = { \setminus. Then }
  basic = { true }
  decl = { alpha(S,R,E,R1,E1,R2,E2) }
  site = { ds(S,R,E) }
  match = \{ () \}
  semantics = { step(E), during(E,R), substep(E1,E), nextAction(E1,E2),
                 during(E1,R1), during(E2,R2) }
  presupposition = { true }
  pragmatics = { elaborate(E) }
  trees = { ds2(S,R,E,R1,E1,R2,E2) }
}.
% Purposive
word = {
```

```
name = \{ \setminus . \}
  basic = { true }
  decl = { alpha(S,T,P,R,E) }
  site = { ds(S,R,E) }
  match = \{ () \}
  semantics = { step(E), during(E,R), substep(P,E), during(P,T) }
  presupposition = { true }
  pragmatics = { purpose(E) }
  trees = { ds3(S,T,P,R,E) }
}.
%% Sentence entries
% Present tense action
word = {
   name = { present }
   basic = { true }
   decl = { alpha(S,R,E,A) }
   site = { s(S,R,E,A) }
   match = \{ () \}
   semantics = { during(E,R), agent(E,A) }
   presupposition = { true }
   pragmatics = { present(R) }
   trees = { simpleS(S,R,E,A) }
}.
% Present tense state
word = {
   name = { present }
   basic = { true }
   decl = { alpha(S,R,configuration(0,P)) }
   site = { s(S,R,configuration(0,P)) }
   match = \{ () \}
   semantics = { configuration(R,O,P) }
   presupposition = { true }
   pragmatics = { true }
   trees = { predS(S,R,configuration(0,P)) }
}.
%% Verbs
word = {
  name = { open }
  basic = { true }
  decl = { alpha(S,R,E,O) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R),
                postcondition(E, configuration(0, state(open))) }
  presupposition = { configuration(R, 0, state(closed)) }
  pragmatics = { true }
```

```
trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { remove }
 basic = { true }
 decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R),
     ?L?X(configuration(R, O, location(L)), locObj(L,X),
          postcondition(E,configuration(0,location(awayFrom(X))))) }
 presupposition = { true }
 pragmatics = { property(0, movable, true) }
 trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { move }
 basic = { true }
  decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R), motion(E, 0, translational) }
 presupposition = { true }
 pragmatics = { property(0, movable, true) }
  trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { turn }
 basic = { true }
  decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R), motion(E, 0, rotational) }
 presupposition = { true }
 pragmatics = { property(0, turnable, true) }
  trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { press }
 basic = { true }
  decl = { alpha(S,R,E,O) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R), force(E,0), pathDir(E,inDir(0)),
                magnitude(E,greater(resistance(0))) }
  presupposition = { true }
 pragmatics = { true }
```

```
trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { hold }
 basic = { true }
 decl = { alpha(S,R,E,O) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R), force(E,0), magnitude(E,weight(0)),
                pathDir(E,oppositeDir(gravity)) }
 presupposition = { true }
 pragmatics = { true }
 trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { normalize }
 basic = { true }
  decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
 match = \{ () \}
  semantics = { during(E,R),
                postcondition(E, configuration(0, state(normal))) }
 presupposition = { true }
 pragmatics = { abstractObj(0) }
  trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { start }
 basic = { true }
  decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { during(E,R),
                postcondition(E, configuration(0, state(running))) }
 presupposition = { true }
 pragmatics = { type(0, pump) }
  trees = { transitiveVP(S,R,E,0) }
}.
word = {
 name = { reset }
  basic = { true }
  decl = { alpha(S,R,E,0) }
  site = { vp(S,R,E) }
 match = \{ () \}
  semantics = { during(E,R),
                postcondition(E, configuration(0, state(reset))) }
 presupposition = { true }
```

```
pragmatics = { true }
  trees = { transitiveVP(S,R,E,0) }
}.
%% Prepositions
word = {
  name = \{ to \}
   basic = { true }
   decl = { beta(S,R,E,P,O) }
   site = { vp(S,R,E) }
   match = \{ () \}
   semantics = { pathEnd(E,P), locObj(P,0) }
   presupposition = { true }
  pragmatics = { true }
   trees = { bVPpp(S,R,E,P,0) }
}.
word = {
  name = { within }
   basic = { true }
   decl = { alpha(R, O, P, I, L) }
   site = { pred(R, 0, P) }
   match = \{ () \}
   semantics = { configuration(R, 0, state(within(L))) }
   presupposition = { true }
   pragmatics = { true }
   trees = { pp(R,O,P,within,L) }
}.
%% Adverbs
word = {
  name = { counterclockwise }
  basic = { true }
  decl = { beta(S,R,E) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { pathDir(E, ccw) }
  presupposition = { true }
 pragmatics = { true }
  trees = { bVPadv(S,R,E) }
}.
word = {
  name = { clockwise }
  basic = { true }
  decl = { beta(S,R,E) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { pathDir(E, cw) }
```

```
presupposition = { true }
  pragmatics = { true }
  trees = { bVPadv(S,R,E) }
}.
%% Nouns
word = {
  name = { pump }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, pump) }
  presupposition = { true }
 pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { panel }
  basic = { true }
  decl = \{ alpha(X) \}
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, panel) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { pressure }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { abstractObj(X), type(X, pressure) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { range }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { abstractObj(X), type(X, range) }
  presupposition = { true }
```

```
pragmatics = { true }
 trees = { aTheNP(X), aANP(X) }
}.
word = {
 name = { position }
 basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { abstractObj(X), type(X, position) }
  presupposition = { true }
 pragmatics = { true }
 trees = { aTheNP(X), aANP(X) }
}.
word = {
 name = { dial }
 basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
 match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, dial) }
 presupposition = { true }
 pragmatics = { true }
 trees = { aTheNP(X), aANP(X) }
}.
word = {
 name = { gauge }
 basic = { true }
 decl = { alpha(X) }
  site = { np(X) }
 match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, gauge) }
 presupposition = { true }
 pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
 name = { light }
 basic = { true }
 decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, light) }
 presupposition = { true }
 pragmatics = { true }
 trees = { aTheNP(X), aANP(X) }
}.
```

```
word = {
  name = { cover }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, cover) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { screw }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, screw) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { lever }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, lever) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
word = {
  name = { button }
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (number singular; person third; gender neuter) }
  semantics = { physicalObj(X), type(X, button) }
  presupposition = { true }
  pragmatics = { true }
  trees = { aTheNP(X), aANP(X) }
}.
```

%% Adjectives/labels

```
word = {
  name = { normal }
   basic = { true }
   decl = { beta(0) }
   site = { n(0) }
   match = \{ () \}
   semantics = { property(0, label, normal) }
   presupposition = { true }
   pragmatics = { true }
   trees = { bAdjN(0) }
}.
word = {
  name = { status }
   basic = { true }
   decl = { beta(0) }
   site = { n(0) }
   match = \{ () \}
   semantics = { property(0, label, status) }
   presupposition = { true }
   pragmatics = { true }
   trees = { bAdjN(0) }
}.
word = {
   name = { RESET }
   basic = { true }
   decl = \{ beta(0) \}
   site = { n(0) }
   match = \{ () \}
   semantics = { property(0, label, reset) }
   presupposition = { true }
   pragmatics = { true }
   trees = { bAdjN(0) }
}.
word = {
   name = { ON }
   basic = { true }
   decl = \{ beta(0) \}
   site = { n(0) }
   match = \{ () \}
   semantics = { property(0, label, on) }
   presupposition = { true }
   pragmatics = { true }
   trees = { bAdjN(0) }
}.
word = {
  name = { loose }
   basic = { true }
```

```
decl = { alpha(R, 0, P) }
   site = { pred(R,0,P) }
   match = \{ () \}
   semantics = { configuration(R, 0, state(loose)) }
   presupposition = { true }
   pragmatics = { true }
   trees = { adj(R,0,P) }
}.
word = {
   name = { green }
   basic = { true }
   decl = { alpha(R, 0, P) }
   site = { pred(R, O, P) }
   match = \{ () \}
   semantics = { configuration(R, 0, color(green)) }
   presupposition = { true }
  pragmatics = { true }
   trees = { adj(R,0,P) }
}.
%% Purpose/means clauses
% Fronted ''to'' purpose clause
word = {
  name = { to }
  basic = { true }
  decl = { beta(S, R, P, E, T, A) }
  site = { s(S,R,P,A) }
  match = \{()\}
  semantics = { subactions(P,E), agent(E,A), during(E,T), starts(T,R) }
  presupposition = { true }
  pragmatics = { command, complex(P) }
  trees = { compSfronted(S,R,P,E,T,A) }
}.
% Means ''by'' clause
word = {
  name = { by }
  basic = { true }
  decl = { beta(S, R, P, E, T) }
  site = { vp(S,R,P) }
  match = \{ () \}
  semantics = { subactions(P,E), during(E,T), starts(T,R) }
  presupposition = { true }
  pragmatics = { true }
  trees = { VPcomp(S,R,P,E,T) }
}.
% Non-fronted ''to'' purpose clause
word = {
```

```
name = { to }
  basic = { true }
  decl = { beta(S,R,E,P,T) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { purpose(E, generate(P)), during(P,T) }
  presupposition = { true }
  pragmatics = { true }
  trees = { compVP(S,R,E,P,T) }
}.
%% Until clause
word = {
  name = { until }
  basic = { true }
  decl = { beta(S,R,E,P,T) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { postcondition(E, P) }
  presupposition = { result(E, T) }
 pragmatics = { true }
  trees = { VPcompS(S,R,E,P,T) }
}.
%% Miscellaneous
% Empty subject for imperative
word = {
  name = \{e\}
  basic = { true }
  decl = { alpha(X) }
  site = { np(X) }
  match = { (case nom) }
  semantics = { hearer(X) }
  presupposition = { true }
  pragmatics = { command }
  trees = { epsilonNP(X) }
}.
% Conjunction of sequential actions
word = {
  name = { and then }
  basic = { true }
  decl = { beta(S,R,E,N,T) }
  site = { vp(S,R,E) }
  match = \{ () \}
  semantics = { nextAction(E,N), during(N,T) }
  presupposition = { true }
  pragmatics = { true }
  trees = { bVPconjoin(S,R,E,N,T) }
```

# A.2 Trees

}.

```
% Initial trees
entry = {
  name = { ds1(S,R,E) }
  pragmatics = { true }
  tree = {
    node = \{
      type = { ds(S,R,E) }
      top = \{ (cat ds) \}
      bottom = { (cat ds) }
      kids = \{
        subst = { type = { s(S,R,E,A) }
                   top = { (cat s) } }
        anchor = \{ index = \{1\} \} \}
    }
  }
}.
entry = {
  name = { ds2(S,R,E,R1,E1,R2,E2) }
  pragmatics = { true }
  tree = {
    node = \{
      type = { ds(S,R,E) }
      top = { (cat ds) }
      bottom = { (cat ds) }
      kids = {
        subst = { type = { s(S,R1,E1,A) }
                   top = \{ (cat s) \} \}
        anchor = \{ index = \{1\} \}
        anchor = \{ index = \{2\} \}
        subst = { type = { s(S,R2,E2,A) }
                   top = \{ (cat s) \} \}
        anchor = \{ index = \{1\} \} \}
    }
  }
}.
entry = {
  name = { ds3(S,T,P,R,E) }
  pragmatics = { true }
  tree = {
    node = \{
      type = { ds(S,R,E) }
      top = \{ (cat ds) \}
      bottom = { (cat ds) }
```

```
kids = {
        subst = { type = { s(S,T,P,A) }
                  top = \{ (cat s) \} \}
        anchor = \{ index = \{1\} \} \}
    }
  }
}.
entry = {
 name = { simpleS(S,R,E,A) }
 pragmatics = { true }
  tree = {
   node = \{
      type = { s(S,R,E,A) }
      top = { (cat s; tense present) }
      bottom = { (cat s; tense present) }
      kids = {
        subst = { type = { u:np(A) }
                  top = { (cat np; number X; person Y; case nom) } }
        node = \{
          type = { ip(S,R,E) }
          top = { (cat ip; form main; tense present; number X; person Y) }
          bottom = { (cat ip; form main; tense present; number X; person Y) }
          kids = {
            node = { type = { u:infl(S,R,E) }
                     top = { (form main; tense present; number X; person Y) }
                     bottom = {(form main; tense present; number X; person Y)}
                     kids = { anchor = { index = \{1\}} }
                   }
            subst = { type = { vp(S,R,E) }
                      top = {(cat vp; tense present; form main;
                              number X; person Y)}}
}}}}.
entry = {
  name = { predS(S,R,configuration(0,P)) }
 pragmatics = { true }
  tree = {
   node = \{
    type = { s(S,R,configuration(0,P)) }
    top = { (cat s; tense present) }
    bottom = { (cat s; tense present) }
    kids = {
      subst = { type = { u:np(0) } }
                top = { (cat np; number X; person Y; case nom) } }
      node = \{
        type = { ip(S,R,P) }
        top = { (cat ip; tense present; form main; number X; person Y) }
        bottom = { (cat ip; tense present; form main; number X; person Y) }
        kids = {
          node = {type = { u:infl(S,R,P) }
```

```
top = { (tense present; form main; number X; person Y) }
                   bottom = { (tense present; form main; number X; person Y) }
                   kids = { anchor = { index = \{1\}\}}
          node = { type = { vp(S,R,P) }
                   top = { (cat vp; tense present; form main;
                            number X; person Y) }
                   bottom = { (cat vp; tense present; form main;
                               number X; person Y) }
                   kids = {
                     node = { type = { v(be) }
                               top = { (cat v; tense present; form main) }
                              bottom = { (cat v; tense present; form main) }
                              kids = { words = { words = {be} } } }
                     subst = { type = {pred(R, 0, P)}
                               top = { (cat pred) } } }
} } } } }.
entrv = {
  name = { transitiveVP(S,R,E,0) }
  pragmatics = { true }
  -
tree = {
    node = \{
      type = { vp(S,R,E) }
      top = { (cat vp; form M; tense T; number X; person Y) }
      bottom = { (cat vp; form M; tense T; number X; person Y) }
      kids = \{
        node = { type = { v(E) }
                 top = { (cat v; form M; tense T; number X; person Y) }
                 bottom = { (cat v; form M; tense T; number X; person Y) }
                 kids = { anchor = { index = \{1\} \} } }
        subst = { type = { u:np(0) }
                  top = { (cat np) } } } } }.
entry = {
name = { p:aTheNP(I) }
pragmatics = { property(I, unique, true) }
tree = {
node = \{
        type = { u:np(I) }
        top = { (cat np; number X; gender Y) }
        bottom = {(cat np; number X; gender Y) }
        kids = {
                words = { words = {the}}
                node = \{
                        type = { p:n(I) }
                        top = {(cat n; number X; gender Y) }
                        bottom = {(cat n; number X; gender Y) }
                        kids = { anchor = { index = \{1\}\}}
                }
        }
}}}.
```

```
entry = {
name = \{ aANP(I) \}
pragmatics = { true }
tree = {
node = \{
        type = { u:np(I) }
        top = { (cat np; number X; gender Y) }
        bottom = {(cat np; number X; gender Y) }
        kids = {
                 words = \{ words = \{a\} \}
                node = \{
                         type = \{ n(I) \}
                         top = {(cat n; number X; gender Y) }
                         bottom = {(cat n; number X; gender Y) }
                         kids = { anchor = {index = \{1\}}}
                 }
        }
}}}.
entry = {
  name = { adj(R, 0, P) }
  pragmatics = { true }
  tree = {
    node = { type = { pred(R,O,P) }
             top = { (cat pred) }
             bottom = { (cat pred) }
             kids = {
              node = { type = { adj(P) }
                        top = { (cat adj) }
                        bottom = { (cat adj) }
                        kids = { anchor = { index = \{1\} \} \} \} \} \}.
entry = {
  name = { pp(R, 0, P, I, L) }
  pragmatics = { true }
  tree = {
   node = { type = { pred(R, 0, P) }
            top = { (cat pred) }
            bottom = { (cat pred) }
            kids = {
             node = { type = { u:pp(P) } }
             top = { (cat pp) }
             bottom = { (cat pp) }
             kids = {
               node = { type = { p(I) }
                         top = \{ (cat p) \}
                         bottom = { (cat p) }
                         kids = { anchor = { index = \{1\} \} } }
               subst = \{ type = \{ u:np(L) \} \}
```
```
top = { (cat np) } } } } } }.
entry = {
  name = { epsilonNP(A) }
  pragmatics = { hearer(A) }
  tree = {
    node = { type = { np(A) }
             top = { (cat np; case nom) }
             bottom = { (cat np; case nom) }
             kids = { anchor = { index = \{1\} \} \} \} }.
% Auxiliary trees
entry = {
  name = { compVP(S,R,E,P,T) }
  pragmatics = { true }
  tree = {
    node = { type = { vp(S,R,E) }
             top = { (cat vp; tense X; form M) }
             bottom = { (cat vp; tense X; form M) }
             kids = {
               foot = { type = { vp(S,R,E) }
                        top = { (cat vp; tense X; form M) } }
               node = { type = { cvp(S,T,P) }
                        top = { (cat cvp; tense X; form M) }
                        bottom = { (cat cvp; tense X; form M) }
                        kids = {
                          node = { type = { comp }
                                   top = \{ (cat comp) \}
                                   bottom = { (cat comp) }
                                   kids = { anchor = { index = \{1\} \} } }
                          subst = { type = { vp(S,T,P) }
                                     top = { (cat vp; tense X; form M) } } } }
}}}.
entry = {
  name = { compSfronted(S,R,P,E,T,A) }
  pragmatics = { true }
  tree = {
    node = \{
      type = { s(S,R,P,A) }
      top = { (cat s; tense X) }
      bottom = { (cat s; tense X) }
      kids = {
        node = { type = { cs(S,R,P,A) }
                 top = { (cat cs; tense X) }
                 bottom = { (cat cs; tense X) }
                 kids = {
                   node = { type = { comp }
                            top = { (cat comp) }
                            bottom = { (cat comp) }
```

```
kids = { anchor = { index = \{1\} \} } }
                   foot = { type = { u:s(S,R,P,A) }
                            top = { (cat s; tense X) } } }
        words = { words = { \backslash, } }
        subst = { type = { s(S,T,E,A) }
                  top = { (cat s; tense X) } } } }.
entry = {
  name = { VPcomp(S,R,P,E,T) }
  pragmatics = { true }
  tree = {
    node = { type = { vp(S,R,P) }
             top = { (cat vp; form M; tense X; number Y; person Z) }
             bottom = { (cat vp; form M; tense X; number Y; person Z) }
             kids = {
               foot = { type = { vp(S,R,P) }
                        top = { (cat vp; form M; tense X; number Y; person Z) } }
               node = { type = { cvp(S,T,E) }
                        top = \{ (cat cvp) \}
                        bottom = { (cat cvp) }
                        kids = {
                          node = { type = { comp }
                                    top = \{ (cat comp) \}
                                   bottom = { (cat comp) }
                                   kids = { anchor = { index = \{1\} \} } }
                          subst = { type = { vp(S,T,E) }
                                     top = { (cat vp; form gerund) } } }
} } } }.
entry = {
  name = { VPcompS(S,R,E,P,T) }
  pragmatics = { true }
  tree = {
    node = { type = { vp(S,R,E) }
             top = { (cat vp; tense X) }
             bottom = { (cat vp; tense X) }
             kids = {
               foot = { type = { vp(S,R,E) }
                        top = { (cat vp; tense X) } }
               node = { type = { cs(S,T,P) }
                        top = { (cat cs; tense X) }
                        bottom = { (cat cs; tense X) }
                        kids = {
                          node = { type = { comp }
                                    top = { (cat comp) }
                                   bottom = { (cat comp) }
                                   kids = { anchor = { index = \{1\} \} } }
                          subst = { type = { s(S,T,P) }
                                     top = { (cat s; tense X) } } } }
} }.
```

```
entry = {
  name = \{ bAdjN(0) \}
  pragmatics = { true }
  tree = {
    node = { type = { n(0) }
             top = \{ (cat n) \}
             bottom = { (cat n) }
             kids = {
               node = { type = { u:adj }
                        top = { (cat adj) }
                        bottom = { (cat adj) }
                        kids = {
                           anchor = \{ index = \{1\} \} \}
               foot = { type = { n(0) }
                        top = { (cat n) } } } } } .
entry = {
name = { bVPpp(S,R,E,P,0) }
pragmatics = {true}
tree = {
        node = \{
        type = { vp(S,R,E) }
        top = { (cat vp; form M; tense T) }
        bottom = { (cat vp; form M; tense T) }
        kids = {
          foot = { type = { vp(S,R,E) }
                   top = { (cat vp; form M; tense T) } }
          node = { type = { pp(P) }
                   top = \{ (cat pp) \}
                   bottom = { (cat pp) }
                   kids = {
                     node = \{ type = \{ p \} \}
                               top = \{ (cat p) \}
                               bottom = { (cat p) }
                              kids = { anchor = { index = {1} } } }
                     subst = { type = { u:np(0) }
                                top = { (cat np; case obj) } } }
}}}.
entry = {
  name = { bVPadv(S,R,E) }
  pragmatics = { true }
  tree = {
    node = { type = { vp(S,R,E) }
             top = { (cat vp; form M; tense T) }
             bottom = { (cat vp; form M; tense T) }
             kids = \{
               foot = { type = { vp(S,R,E) }
                        top = { (cat vp; form M; tense T) } }
               node = { type = { adv }
```

```
top = \{ (cat adv) \}
                        bottom = { (cat adv) }
                        kids = { anchor = { index = \{1\}} }
               }
             }
   }
  }
}.
entry = {
  name = { bVPconjoin(S,R,E,N,T) }
  pragmatics = { true }
  tree = {
    node = { type = { vp(S,R,E) }
             top = { (cat vp; form M; tense X) }
             bottom = { (cat vp; form M; tense X) }
             kids = {
               foot = { type = { vp(S,R,E) }
                        top = { (cat vp; form M; tense X) } }
               anchor = \{ index = \{1\} \}
               anchor = \{ index = \{2\} \}
               subst = { type = { vp(S,T,N) }
                         top = { (cat vp; form M; tense X) } } } }.
```

## A.3 Modal operators

dim local.

% G has information used to test specificities G S4. % C is overall general common knowledge C S4 G. U1 S4 C. U2 S4 U1. U3 S4 U2. P1 S4 U1. P2 S4 U2. P3 S4 U3. S S4 P1 P2 P3.

## A.4 Domain knowledge

% KNOWLEDGE FOR THE PANEL DOMAIN EXAMPLES

```
G true.
% Knowledge about the general situation
C hearer(u).
C domain(u,u).
C command.
C *P *R (subinterval(R,P) -> C(present(P) -> present(R))).
*P *R C(starts(R,P) -> subinterval(R,P)).
*P *R1 *R2 C(meets(R2,R1), subinterval(R1,P) -> subinterval(R2,P)).
% Knowledge about objects
% Rules for reasoning about objects
*0 C (type(0, standardScrew) -> type(0, screw)).
*O C (type(O, controlPanel) -> type(O, panel)).
*O C (type(O, standardScrewHole) -> type(O, hole)).
*O C (type(0, controlLever) -> type(0, lever)).
*0 C (type(0, standardButton) -> type(0, button)).
*0 ((physicalObj(0) ; abstractObj(0)) -> object(0)).
% Object distractor rule
*01 *02 (object(01), object(02) -> C domain(01,02)).
% Object specifications
C physicalObj(pump1).
C type(pump1, pump).
C hasPart(pump1, pressure1).
C hasPart(pump1, panel1).
C property(pump1, state, oneOf(running, reset, halted)).
C property(pump1, controlSource, list(lever1, button1, physicalModel)).
C property(pump1, movable, true).
C property(pump1, unique, true).
C abstractObj(pressure1).
C type(pressure1, pressure).
C partOf(pressure1, pump1).
C property(pressure1, controlSource, list(dial1, physicalModel)).
C property(pressure1, unique, true).
C physicalObj(panel1).
C type(panel1, controlPanel).
C partOf(panel1, pump1).
C hasPart(panel1, cover1).
```

```
C hasPart(panel1, gauge1).
C hasPart(panel1, dial1).
C hasPart(panel1, lever1).
C hasPart(panel1, light1).
C hasPart(panel1, button1).
C property(panel1, movable, false).
C property(panel1, unique, true).
C physicalObj(cover1).
C type(cover1, cover).
C partOf(cover1, panel1).
C hasPart(cover1, screw1).
C hasPart(cover1, hole1).
C property(cover1, defaultLocation, on(panel1)).
C property(cover1, movable, true).
C property(cover1, unique, true).
C physicalObj(screw1).
C type(screw1, standardScrew).
C partOf(cover1).
C property(screw1, state, oneOf(tight,loose)).
C property(screw1, loosenDir, ccw).
C property(screw1, tightenDir, cw).
C property(screw1, defaultLocation, in(hole1)).
C property(screw1, turnable, true).
C property(screw1, movable, true).
C property(screw1, unique, true).
C physicalObj(hole1).
C type(hole1, standardScrewHole).
C partOf(hole1, cover1).
C property(hole1, location, in(cover1)).
C property(hole1, movable, false).
C property(hole1, unique, true).
C physicalObj(gauge1).
C type(gauge1, gauge).
C partOf(gauge1, panel1).
C hasPart(gauge1, range1).
C hasPart(gauge1, range2).
C property(gauge1, controlSource, pressure1).
C property(gauge1, state, oneOf(within(range1),outside(range1))).
C property(gauge1, movable, false).
C property(gauge1, unique, true).
C abstractObj(range1).
C type(range1, range).
C partOf(range1, gauge1).
C property(range1, label, normal).
C property(range1, unique, true).
```

```
C abstractObj(range2).
C type(range2, range).
C partOf(range2, gauge1).
C property(range2, unique, true).
C physicalObj(dial1).
C type(dial1, dial).
C partOf(dial1, panel1).
C property(dial1, turnDir, cw).
C property(dial1, controls, pressure1).
C property(dial1, turnable, true).
C property(dial1, unique, true).
C physicalObj(lever1).
C type(lever1, controlLever).
C partOf(lever1, panel1).
C hasPart(lever1, resetPos1).
C hasPart(lever1, defaultPos1).
C hasPart(lever1, leverSlot1).
C property(lever1, weight, fixed).
C property(lever1, defaultLocation, at(defaultPos1)).
C property(lever1, returns2defaultLocation, true).
C property(lever1, controls, pump1).
C property(lever1, movable, true).
C property(lever1, unique, true).
C abstractObj(resetPos1).
C type(resetPos1, position).
C partOf(resetPos1, lever1).
C property(resetPos1, label, reset).
C property(resetPos1, location, above(defaultPos1)).
C property(resetPos1, movable, false).
C property(resetPos1, unique, true).
C abstractObj(defaultPos1).
C type(defaultPos1, position).
C partOf(defaultPos1, lever1).
C property(defaultPos1, location, bottom(leverSlot1)).
C property(defaultPos1, movable, false).
C property(defaultPos1, unique, true).
C physicalObj(leverSlot1).
C type(leverSlot1, slot).
C partOf(leverSlot1, lever1).
C property(leverSlot1, movable, false).
C property(leverSlot1, unique, true).
C physicalObj(light1).
C type(light1, light).
C partOf(light1, panel1).
C property(light1, label, status).
```

```
C property(light1, color, oneOf(red, green)).
C property(light1, controlSource, pump1).
C property(light1, movable, false).
C property(light1, unique, true).
C physicalObj(button1).
C type(button1, standardButton).
C partOf(button1, panel1).
C property(button1, controls, pump1).
C property(button1, resistance, fixed).
C property(button1, inDir, prependicularTo(panel1)).
C property(button1, label, on).
C property(button1, movable, true).
C property(button1, unique, true).
C physicalObj(genericButton).
C type(genericButton, standardButton).
% Initial state of objects
C configuration(t_1, panel1, state(closed)).
C configuration(t_1, cover1, location(on(panel1))).
C configuration(t_1, screw1, location(in(hole1))).
*T *0 *P *S C((starts(S, T); meets(S,T)), configuration(T, 0, P) ->
                  C configuration(S, O, P)).
% Knowledge about locations
*P *Q (location(P), location(Q) \rightarrow C domain(P, Q)).
*O C location(at(O)).
*0 C locObj(at(0),0).
*0 C locObj(location(at(0)),0).
*0 C location(on(0)).
*0 C locObj(on(0),0).
*O C locObj(location(on(O)),O).
*0 C location(awayFrom(0)).
*O C locObj(awayFrom(O),O).
*O C locObj(location(awayFrom(O)),O).
*O C location(in(O)).
*0 C locObj(in(0),0).
*O C locObj(location(in(O)),O).
*0 C location(over(0)).
*0 C locObj(over(0),0).
*O C locObj(location(over(O)),O).
% Knowledge about action termination
```

% "If a postcondition is part of the action information, then the % action has termination information."

```
*E *P C(postcondition(E,P) -> termination(E)).
\% "If an action has a bounded path, then the action has termination."
*E *O *T *P C(motion(E,O,T), pathEnd(E, P) -> termination(E)).
\% "If an action has a purpose and the purpose has a termination, then
% the action has termination."
*A *P C(purpose(A,generate(P)) -> termination(A)).
% Knowledge about actions in the domain
\% General knowledge about actions and what it means for actions to be
% "concrete"
*A *R *O *P C(postcondition(A, configuration(O, P)), result(A, R) ->
                configuration(R,O,P)).
*A C(pathDir(A,D) -> path(A)).
*A C(pathEnd(A,D) -> path(A)).
*A *O *T C(motion(A,O,T), path(A) -> concrete(A)).
*A *O *W *D C(force(A,O), magnitude(A,W), pathDir(A,D) -> concrete(A)).
*A *O *T *P U2(motion(A,O,T), purpose(A,P) -> concrete(A)).
*A *O *T *P U2(force(A,O), purpose(A,P) -> concrete(A)).
*P C(purpose(P), (?A purpose(A,generate(P))) -> concrete(P)).
*A C((?S subactions(A, S), concreteAll(S)) -> concrete(A)).
C concreteAll(nil).
*A C((?N nextAction(A, N), concrete(A), concreteAll(N)) -> concreteAll(A)).
*S *A U1(type(S,screw),
         postcondition(A, configuration(S, state(loose))) ->
             motion(A,S,rotational),
            (?D property(S, loosenDir, D), pathDir(A,D))).
% Knowledge for opening the panel
*A (postcondition(A, configuration(panel1, state(open))) ->
     U3 ?S(subactions(A,S), nextAction(S,nil),
           postcondition(S, configuration(screw1,location(awayFrom(hole1))))).
*A (postcondition(A, configuration(screw1, location(awayFrom(hole1))))
        -> U2 ?S?N (subactions(A,S),
                    postcondition(S,configuration(screw1,state(loose))),
                    nextAction(S,N), motion(N,screw1,translational),
                    pathEnd(N,awayFrom(hole1)), nextAction(N,nil))).
*A *O (postcondition(A, configuration(O, state(open))) ->
```

```
175
```

```
P1 complex(A)).
*A *O (type(0,screw), motion(A,0,rotational) ->
          U1 instrument(A, screwdriver)).
% Knowledge for normalizing the pressure
*A (postcondition(A,configuration(pressure1,state(normal))) ->
       U3 ?S(subactions(A,S), nextAction(S,nil),
             postcondition(S, configuration(gauge1,state(within(range1)))))).
*A (postcondition(A, configuration(pressure1, state(normal))) ->
       P2 purpose(A)).
*A (postcondition(A,configuration(gauge1,state(within(range1)))) ->
       U2(motion(A, dial1, rotational),
          ?D(property(gauge1,turnDir,D), pathDir(A,D)))).
*A (postcondition(A, configuration(pressure1, state(normal))) ->
       P1 complex(A)).
% Knowledge for starting the pump
*A (postcondition(A, configuration(pump1, state(running))) ->
      U3 ?S?N(subactions(A,S),
              postcondition(S,configuration(pump1,state(reset))),
              nextAction(S,N), nextAction(N,nil),
              force(N,button1), magnitude(N,greater(resistance(button1))),
              pathDir(N,inDir(button1)))).
*A (postcondition(A, configuration(pump1, state(running))) ->
       P2 complex(A)).
*A (postcondition(A, configuration(pump1, state(reset))) ->
      U2 ?S?N(subactions(A,S),
              motion(S,lever1,translational), pathEnd(S,at(resetPos1)),
              nextAction(S,N), force(N,lever1), pathEnd(N,at(resetPos1)),
              pathDir(N, oppositeDir(gravity)), magnitude(N,weight(lever1)),
              postcondition(N,configuration(light1,color(green))))).
*A (postcondition(A, configuration(pump1, state(running))) ->
      P1 elaborate(A)).
% Action: a
C domain(a,a).
S during(a, t).
C result(a, r).
S meets(r, t).
S agent(a, u).
```

```
S postcondition(a, configuration(status1, pump(running))).
S subactions(a, a1).
S nextAction(a, nil).
S parent(a, nil).
% Action: a1
C domain(a1,a1).
S step(a1).
S during(a1, t_1).
S starts(t_1, t).
C result(a1, t_2).
S meets(t_2, t_1).
S agent(a1, u).
S postcondition(a1, configuration(panel1, state(open))).
S subactions(a1, oa1).
S nextAction(a1, a2).
S parent(a1, a).
% Action: oa1
C domain(oa1,oa1).
C domain(t_{1}, t_{1}).
C domain(t_{1}_{2}, t_{1}_{2}).
C time(t_1_1).
S substep(oa1,a1).
S during(oa1, t_1_1).
S starts(t_1_1, t_1).
C result(oa1, t_1_2).
S meets(t_1_2, t_1_1).
S agent(oa1, u).
S postcondition(oa1, configuration(screw1, location(awayFrom(hole1)))).
S subactions(oa1, ua1).
S nextAction(oa1, oa2).
S parent(oa1, a1).
% Action: ual
C domain(ua1,ua1).
C domain(t_1_1, t_1_1, t_1_1).
C domain(t_1_1_2, t_1_1_2).
C time(t_1_1_1).
S during(ua1, t_1_1).
S starts(t_1_1_1, t_1_1).
C result(ua1, t_1_2).
S meets(t_1_1_2, t_1_1_1).
```

```
S agent(ua1, u).
S instrument(ual, screwdriver).
S motion(ua1, screw1, rotational).
S pathDir(ua1, ccw).
S postcondition(ua1, configuration(screw1, state(loose))).
S nextAction(ua1, ua2).
S prevAction(ua1,nil).
S parent(ua1, oa1).
% Action: ua2
C domain(ua2,ua2).
S during(ua2, t_1_2).
S finishes(t_1_1_2, t_1_1).
C result(ua2, t_1_2).
S agent(ua2, u).
S motion(ua2, screw1, translational).
S pathEnd(ua2, awayFrom(hole1)).
S nextAction(ua2, nil).
S prevAction(ua2, ua1).
S parent(ua2, oa1).
% Action: oa2
C domain(oa2,oa2).
S precondition(oa2, oa1).
S during(oa2, t_1_2).
S finishes(t_1_2, t_1).
C result(oa2, t_2).
S agent(oa2, u).
S postcondition(oa2, configuration(cover1, location(awayFrom(panel1)))).
S nextAction(oa2, nil).
S prevAction(oa2, oa1).
S parent(oa2, a1).
% Action: a2
C domain(a2, a2).
C domain(t_3, t_3).
S step(a2).
S during(a2, t_2).
C result(a2, t_3).
S meets(t_3, t_2).
S agent(a2, u).
S postcondition(a2, configuration(pressure1, state(normal))).
S subactions(a2, na1).
```

```
S nextAction(a2, a3).
S prevAction(a2, a1).
S parent(a2, a).
% Action: na1
C domain(na1,na1).
S substep(na1,a2).
S during (na1, t_2_1).
S starts(t_2_1, t_2).
S finishes(t_2_1, t_2).
C result(na1, t_3).
S agent(na1, u).
S motion(na1, dial1, rotational).
S pathDir(na1, cw).
S postcondition(na1, configuration(gauge1, state(within(range1)))).
S purpose(na1, generate(a2)).
S nextAction(na1, nil).
S parent(na1, a2).
% Action: a3
C domain(a3,a3).
S step(a3).
S during (a3, t_3).
C result(a3, t_4).
S meets(t_4, t_3).
S agent(a3, u).
S postcondition(a3, configuration(pump1, state(running))).
S subactions(a3, sa1).
S nextAction(a3, nil).
S prevAction(a3, a2).
S parent(a3, a).
% Action: sa1
C domain(sa1,sa1).
S substep(sa1,a3).
S during(sa1, t_3_1).
S starts(t_3_1, t_3).
C result(sa1, t_3_2).
S meets(t_3_2, t_3_1).
S agent(sa1, u).
S postcondition(sa1, configuration(pump1, state(reset))).
S subactions(sa1, ra1).
```

```
S nextAction(sa1, sa2).
S parent(sa1, a3).
% Action: ra1
C domain(ra1,ra1).
S during(ra1, t_3_1_1).
S \text{ starts}(t_3_1_1, t_3_1).
C result(ra1, t_3_1_2).
S meets(t_3_1_2, t_3_1_1).
S agent(ra1, u).
S motion(ra1, lever1, translational).
S pathEnd(ra1, at(resetPos1)).
S nextAction(ra1, ra2).
S parent(ra1, sa1).
% Action: ra2
C domain(ra2,ra2).
S during(ra2, t_3_1_2).
C result(ra2, t_3_2).
S finishes(t_3_1_2, t_3_1).
S agent(ra2, u).
S force(ra2, lever1).
S magnitude(ra2, weight(lever1)).
S pathDir(ra2, oppositeDir(gravity)).
S postcondition(ra2, configuration(light1, color(green))).
S nextAction(ra2, nil).
S prevAction(ra2, ra1).
S parent(ra2, sa1).
% Action: sa2
C domain(sa2,sa2).
S during(sa2, t_3_2).
S finishes(t_3_2, t_3).
C result(sa2, t_4).
S agent(sa2, u).
S force(sa2, button1).
S magnitude(sa2, greater(resistance(button1))).
S pathDir(sa2, inDir(button1)).
S purpose(sa2, generate(a3)).
S nextAction(sa2, nil).
S prevAction(sa2, sa1).
```

```
S parent(sa2, a3).
```

## A.5 Morphological knowledge

```
present = begin
        (form main; mode interrogative; number singular; person third) ~> does ;
        (form main; mode interrogative) ~> do ;
        () ~> ;
end.
you = begin () ~> you ; end.
e = begin (cat np; case nom) ~> ; end.
the = begin () \sim; end.
a = begin () \sim a ; end.
from = begin () ~> from ; end.
to = begin () \sim to ; end.
by = begin () \sim by ; end.
in = begin () \sim in ; end.
within = begin () ~> within ; end.
on = begin () \sim on ; end.
onto = begin () ~> onto ; end.
outof = begin () ~> out of ; end.
awayfrom = begin () ~> away from ; end.
counterclockwise = begin () ~> counterclockwise ; end.
clockwise = begin () ~> clockwise ; end.
until = begin () ~> until ; end.
and = begin () \sim and ; end.
then = begin () \sim then ; end.
Then = begin () \sim Then ; end.
? = begin () ~> ? ; end.
, = begin () \sim , ; end.
; = begin () ~> ; ; end.
: = begin () ~> : ; end.
\ = begin () \sim \ ; end.
pump = begin (cat n) ~> pump ; end.
panel = begin (cat n) ~> panel ; end.
cover = begin (cat n) ~> cover ; end.
dial = begin (cat n) ~> dial ; end.
lever = begin (cat n) ~> lever ; end.
gauge = begin (cat n) ~> gauge ; end.
pressure = begin (cat n) ~> pressure ; end.
position = begin (cat n) ~> position ; end.
light = begin (cat n) ~> light ; end.
hole = begin (cat n) ~> hole ; end.
screw = begin (cat n) ~> screw ; end.
screwdriver = begin (cat n) ~> screwdriver ; end.
button = begin (cat n) ~> button ; end.
```

```
range = begin (cat n) ~> range ; end.
red = begin (cat adj) ~> red ; end.
green = begin (cat adj) ~> green ; end.
loose = begin (cat adj) ~> loose ; end.
tight = begin (cat adj) ~> tight ; end.
normal = begin (cat adj) ~> normal ; end.
RESET = begin (cat adj) ~> RESET ; end.
ON = begin (cat adj) \sim ON ; end.
move = begin
        (cat v; tense present; form main) ~> move ;
        (cat v; form gerund) ~> moving ;
end.
remove = begin
        (cat v; form main; tense present) ~> remove ;
        (cat v; form gerund) ~> removing ;
end.
turn = begin
        (cat v; tense present; form main) ~> turn ;
        (cat v; form gerund) ~> turning ;
end.
open = begin
        (cat v; tense present; form main) ~> open ;
        (cat v; form gerund) ~> opening ;
end.
normalize = begin
        (cat v; tense present; form main) ~> normalize ;
        (cat v; form gerund) ~> normalizing ;
end.
hold = begin
        (cat v; tense present; form main) ~> hold ;
        (cat v; form gerund) ~> holding ;
end.
press = begin
        (cat v; tense present; form main) ~> press ;
        (cat v; form gerund) ~> pressing ;
end.
start = begin
        (cat v; tense present; form main) ~> start ;
        (cat v; form gerund) ~> starting ;
end.
reset = begin
```

```
(cat v; tense present; form main) ~> reset ;
      (cat v; form gerund) ~> resetting ;
end.
be = begin
      (cat v; tense present; form main) ~> is ;
      (cat v; form gerund) ~> being ;
end.
```

## A.6 Generation instances

```
gen = \{
  name = { Step 1 for Novice }
  private = \{ S \}
  shared = { P1 (P1 present(t_1) -> $) }
  describe = { ds(s,t_1,a1) }
  pattern = { ds(S,R,E) }
  features = \{()\}
  communicate = { concrete(a1) termination(a1)
                  concrete(oa1) termination(oa1)
                  concrete(ua1) termination(ua1) nextAction(oa1,oa2)
                  concrete(oa2) termination(oa2) }
}.
gen = \{
  name = { Step 1 for Beginner }
  private = \{ S \}
  shared = { P2 (P2 present(t_1) \rightarrow $) }
  describe = { ds(s,t_1,a1) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { concrete(a1) termination(a1) }
}.
gen = {
  name = { Step 1 for Advanced }
  private = { S $ }
  shared = { P3 (P3 present(t_1) -> $) }
  describe = { ds(s,t_1,a1) }
  pattern = { ds(S,R,E) }
  features = \{()\}
  communicate = { concrete(a1) termination(a1) }
}.
gen = \{
  name = { Step 2 for Novice }
  private = { S $ }
  shared = { P1 (P1 present(t_2) -> $) }
  describe = { ds(s,t_2,a_2) }
```

```
pattern = { ds(S,R,E) }
  features = \{()\}
  communicate = { concrete(a2) termination(a2)
                  concrete(na1) termination(na1) }
}.
gen = \{
  name = { Step 2 for Beginner }
  private = { S $ }
  shared = { P2 (P2 present(t_2) -> $) }
  describe = { ds(s,t_2,a2) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { concrete(a2) termination(a2) }
}.
gen = \{
  name = { Step 2 for Advanced }
  private = { S $ }
  shared = { P3 (P3 present(t_2) -> $) }
  describe = { ds(s,t_2,a_2) }
  pattern = { ds(S,R,E) }
  features = \{()\}
  communicate = { concrete(a2) termination(a2) }
}.
gen = \{
  name = { Step 3 for Novice }
  private = \{ S \}
  shared = { P1 (P1 present(t_3) -> $) }
  describe = { ds(s,t_3,a3) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { subactions(sa1,ra1) nextAction(sa1,sa2)
                  concrete(ra1) termination(ra1) nextAction(ra1,ra2)
                  concrete(ra2) termination(ra2)
                  purpose(sa2,generate(a3)) }
}.
gen = \{
  name = { Step 3 for Beginner }
  private = \{ S \}
  shared = { P2 (P2 present(t_3) -> $) }
  describe = { ds(s,t_3,a3) }
  pattern = { ds(S,R,E) }
  features = \{ () \}
  communicate = { concrete(a3) termination(a3)
                  concrete(sa1) termination(sa1) nextAction(sa1,sa2) }
}.
```

```
gen = \{
```

```
name = { Step 3 for Advanced }
private = { S $ }
shared = { P3 (P3 present(t_3) -> $) }
describe = { ds(s,t_3,a3) }
pattern = { ds(S,R,E) }
features = { () }
communicate = { concrete(a3) termination(a3) }
}.
```

# Appendix B Excerpts from coded corpus

These appendix presents randomly selected excerpts from the three sources of coded corpus data.

## B.1 [Reader's Digest, 1991]

Excerpts from "Reader's Digest New Complete Do-It-Yourself Manual". Published by The Reader's Digest Association, Pleasantville, NY, 1991. Scanned by Joseph Rosenzweig, March 1994.

#### Chapter One: Emergency repairs in your home

#### Draining the system

1.	<b>Turn</b> off the main water supply valve.	$\mathbf{AC}[\mathbf{p}]$
2.	<b>Stop</b> the water supply to the water heater $by$ closing value on pipe leadi $IC$ [pc-by:IC]	ng into heater.
	If you have a gas heater, $\mathbf{turn} \ off$ the gas cock.	$\mathbf{AC}[\mathbf{p}]$
	If you have an electric water heater, <b>switch</b> off its circuit breaker or <b>re</b> that controls heater's circuit.	$\mathbf{move}$ the fuse $\mathbf{IC}[\mathbf{p}],  \mathbf{IC}$
3.	If your heating system utilizes a boiler, $\mathbf{shut}$ off water supply line the water to the boiler (the valve should be near the boiler on pipe leading into it	1  ater inlet valve ). $\mathbf{IC}[\mathbf{p}]$
	Then <b>flush</b> all the toilets in the house and <b>open</b> all faucets.	IC, IC
4.	If house is heated by a hot-water system, <b>open</b> the valves on all radiator individual valves).	ts (if they have $\mathbf{IC}$
	Then <b>open</b> the air vents on one or more radiators (baseboard or other ty floor of house.	${ m pe}$ ) on highest ${ m IC}$
	Hold cup under vent and <b>catch</b> water as it spurts out.	$\mathbf{AT}[\mathrm{oa}],  \mathbf{IC}$
5.	Let the water in the boiler $\mathit{cool}$ ( $\mathbf{check}$ the temperature indicator $\mathbf{AC}[\mathrm{arg}],\mathbf{IC}$	on the unit ).
	$\mathbf{Attach}$ a hose to the drain valve near the base of the boiler and $\mathbf{lead}$ it	outdoors or $to$
	a drain lower than the boiler. IO	C, $\mathbf{AC}[adv, pp]$
	<b>Open</b> the outlet and <b>let</b> the water <i>flow out</i> .	$\mathbf{IC},\ \mathbf{AC}[\mathrm{arg}]$
	186	

6.	Attach a hose to the draincock of the water heater and <b>direct</b> the hose to a place lower than the heater's draincock or <i>outdoors</i> and <i>away</i> from the AC[pp,adv]	<i>into</i> a drain, e house. <b>IC</b> ,
	<b>Open</b> the drain valve and <b>let</b> the water <i>run out</i> .	IC, AC[arg]
7.	<b>Open</b> draincock on main water supply line.	IC
	If no such spigot exists, <b>disconnect</b> a fitting at lowest point in system $t$ of water to run out. $IC[pc$	o <b>allow</b> rest -to: <b>AC</b> [arg]]
	If your water comes from a well, <b>switch</b> pump circuit <i>off</i> ; <b>drain</b> above-g lines and the storage tank.	round pump IC[p], IC
8.	<b>Empty</b> toilet bowls and tanks $by$ siphoning or bailing and sponging. by: $AC[oa]]$	IC[pc-
	<b>Pour</b> propylene glycol antifreeze (from plumbing supplier or RV or marine toilet bowls, sinks, tubs, dishwasher, and washing machine.	dealer) into AC[arg,pp]
	When power returns, $\mathbf{put}$ both washers through cycle.	IC,
Chapter :	Two: Hand tools and how to use them	
Hand too	ls:Constructing a workbench 1. Measure and cut legs.	IC, AC[arg]
	<b>Subtract</b> thickness of plywood layers (step 6) from final height <i>to</i> <b>determ</b> of long leg pieces.	mine length IC[pc-to:IC]
	$\mathbf{Cut}$ other pieces 3 1/2 in. shorter than long pieces.	$\mathbf{AC}[\mathrm{arg}]$
	<b>Glue</b> and <b>nail</b> one long and one short piece to form a unit. $AC[oa]$ , A	<b>\C</b> [pc-to:IC]
	Install nails in zigzag pattern.	IC
2.	$\mathbf{Cut}$ two cross braces 20 in. long.	$\mathbf{AC}[\mathrm{arg}]$
	Set them in place across short leg pieces.	IC
	<b>Drill</b> 1/4-indiameter holes <i>through</i> braces and legs, two holes per leg. De <b>AC</b> [pp]	o not fasten.
	<b>Label</b> braces and legs for reassembly, and <b>set</b> the braces aside.	IC, IC
3.	$\mathbf{Cut}$ two top rails 45 in. long and two bottom rails 48 in. long.	$\mathbf{AC}[\mathrm{arg}]$
	Align top rails with long leg pieces; bottom rails 8 in. from floor.	IC
	Be sure labeled legs match and rails are inside them.	IC
	Drill two 3/16-in. holes through each connection.	AC[pp]
	Fasten with lag screws and flat washers.	IC
4.	Assemble base.	IC
	Put cross braces in place on legs.	
	<b>Fasten</b> with carriage bolts, nat washers, lock washers, and nuts.	
	<b>Cut</b> shelf, in place on lower rails, and secure it with 8d pails or $1.1/4$ in $x$	
	IC, IC	voou screws.
5.	Cut remaining plywood panel $in$ half lengthwise; then trim panels $to$ leng $\mathbf{AC}[pp]$	,th. <b>AC</b> [pp],
	If mounting a wood working vise, <b>allow</b> a 15-in. overhang at vise end, 4 end, and 2 in. at sides.	in. at other $\mathbf{AC}[\mathrm{arg}]$
	Attach one panel to base with 4d ringed nails or countersunk screws.	IC

6.	<b>Glue</b> panels <i>together</i> with white or yellow glue.	$\mathbf{AC}[adv]$
	<b>Clamp</b> around perimeter, and <b>weight</b> the center to ensure proper bondi $AC$ [pc-to:IC]	ng. <b>IC</b> [pp],
	For extra strength, <b>install</b> countersunk ringed nails or screws around perimintervals.	neter at 1-ft IC
	Cut notch for vise.	$\mathbf{AC}[arg, for]$
7.	Cut 1 x 2's for edging.	$\mathbf{AC}[\mathrm{for}]$
	Tack them <i>around</i> top rim, flush with bench top, with small ringed nails.	$\mathbf{AC}[pp]$
	Butt join (p.100) or miter (p.108) corners.	IC, IC
	Fit vise to workbench, following manufacturer's instructions.	IC
8.	$\mathbf{Cut}$ hardboard $to$ same size as edged bench top.	$\mathbf{AC}[pp]$
	$\mathbf{Apply}$ glue to hardboard and to bench.	$\mathbf{AC}[pp]$
	<b>Align</b> all edges; then <b>apply</b> clamps and weights <i>to</i> <b>maintain</b> position <i>unt</i> <b>IC</b> , <b>AT</b> [pc-to: <b>AT</b> [until]]	<i>il</i> glue dries.
	<b>Finish</b> with at least three coats of polyure hane varnish $(p.121)$ .	IC

## Chapter Three: Power tools and how to use them

## Basic drilling

1.	Insert a bit <i>fully</i> into the chuck. IC[adv]
	Unless your drill has a power-driven automatic chuck lock, <b>turn</b> the chuck key in all three holes <i>so that</i> all the jaws make contact with the bit. $AC$ [pc-st]
2.	$\mathbf{Make\ sure}$ that the piece you are working on is firmly supported or clamped down. $\mathbf{IC}$
	If possible, ${\bf arrange}$ the work $so~that$ you are drilling straight down or straight ahead. ${\bf IC}[{\rm ms}]$
	$\mathbf{Make} \text{ a starter hole with an awl or nail so that the drill bit won't wander.} \qquad \mathbf{IC}[\mathrm{ms}]$
3.	$\begin{array}{llllllllllllllllllllllllllllllllllll$
	<b>Increase</b> speed after the bit has penetrated the surface. <b>AC</b> [oa]
	<b>Push</b> firmly, but don't force the drill to cut too fast. <b>AT</b> [oa]
4.	To help you keep the drill straight, position or clamp a try square or combination square near the drill and keep the drill parallel to the square. $IC[pc-to:AC[oa]], AC[oa]$
5.	When drilling all the way through a piece of wood, <b>clamp</b> a piece of scrap wood behind it <i>to</i> <b>prevent</b> it from splintering. <b>IC</b> [pc-to:IC]
	Or drill <i>into</i> the wood only <i>until</i> the point of the bit emerges, then <b>complete</b> the drilling from the other side. $AC[pp,until]$ , $IC$
6.	$\begin{array}{ll} \textit{To} \mbox{ make a hole of the depth you want, use a commercial drill stop or gauge, or wrap a piece of masking tape at the appropriate height on the bit. & AC[pc-to:IC], \\ IC[pc-to:IC] \end{array}$

Then drill until the stop, gauge, or tape touches the surface of the material being drilled. AT[until]

Adjustments

7.	Put 1/2-in. bit into chuck.	$\mathbf{IC}$
	$\mathbf{Place}$ square in front of bit, then at side, to $\mathbf{check}$ that it is performed as the square of the square o	erpendicular to drill
	press table.	IC[pc-to:IC]
	Adjust, if needed.	$\mathbf{AC}[\mathrm{oa}]$
8.	Rotate same bit close to block of wood.	$\mathbf{AT}[\mathrm{if}]$
	If you see a wobble in gap between bit and wood, $\mathbf{rechuck}$ bit and	${\bf test}$ again. IC, IC
	<b>Replace</b> chuck if wobble persists.	IC
9.	Check angle of table.	IC
	Adjust temporarily by inserting paper or foil shims, as shown.	$\mathbf{AC}[\text{pc-by:IC}]$
	Shim here to tilt downward.	$\mathbf{IC}[\text{pc-to:}\mathbf{AC}[\text{oa}]]$

#### Chapter Four: Fasteners and adhesives

#### Joining with wood screws

1.	<b>Clamp</b> the pieces <i>together</i> .	$\mathbf{IC}[adv]$
	Mark the screw positions and <b>select</b> a drill bit equal to the diameter of	the screw's
	shank (see chart, facing page).	IC, IC
	Mark the top piece's thickness on the bit with tape (or use a drill stop).	$\mathbf{IC}$
	Then <b>drill</b> a shank hole, <b>stopping</b> at tape.	$\mathbf{AC}[\mathrm{fa:IC}]$
2.	${\bf Select}$ a drill bit equal to the screw's diameter minus the threads.	IC
	With a piece of tape, <b>mark</b> the screw's length on the bit.	$\mathbf{IC}$
	<b>Drill</b> pilot hole, <b>stopping</b> at the tape.	$\mathbf{AC}[\mathrm{fa:IC}]$
3.	If you are using flathead screws, <b>drill</b> a countersink hole of the same diam screwhead.	eter as the $\mathbf{AC}[\mathrm{arg}]$
	<b>Check</b> diameter $by$ holding screwhead upside down over hole. IC[pc-	$by: \mathbf{AT}[oa]]$
4.	$\mathbf{Rub}$ wax on screw threads for easier installation.	$\mathbf{AC}[\mathbf{p}]$
	<b>Insert</b> screw in the hole and <b>drive</b> it <i>in until</i> screwhead is flush with surfa <b>IC</b> , <b>AC</b> [p,until]	ce of work.

#### Chapter Five: Woodworking, types of wood, techniques and finishes

#### **Overlapping joints**

- A table saw makes quick work of cutting end laps. Adjust saw blade height so that the teeth just touch the scribed cheek line. AC[pc-st]
   Using a miter gauge to guide work across saw table, make shoulder cut. IC[fa:AT[pc-to:AC[pp]]]
   C. this is the provide the scribe of th
- Cut joint face with a tenoning jig.
   Clamp work vertically in jig and remove waste in a single cut.
   IC, IC
   If you don't have a jig, hold stock as for shoulder cut and remove waste in several passes.
   AC[arg]
   AC[arg]
   IC, IC
   If you don't have a jig, hold stock as for shoulder cut and remove waste in several passes.
- 3. A router with a straight bit can cut several lap joints at the same time. Mark and align the pieces. IC, IC

Clamp a guide board across them, allowing for distance between bit and edge of base plate, so that bit is set for shoulder cut. IC[fa:AT[oa],ms]

4. Rout the waste, beginning at the tips of the pieces and cutting progressively closer to guide board. IC[fa:IC,fa:AC[arg]]
 If you are not experienced with a router, reposition the board to guide each cut. IC[pc-to:AC[arg]]

#### Chapter Six: Metals and plastics, how to work with them

#### Starter holes

- 1. Clamp work securely in a vise, and drill a hole slightly smaller than the diameter of the tap. IC, AC[arg]  $\mathbf{IC}$ (**Check** table at left for size the hole should be.) Lubricate threads of tap with cutting fluid or (better still) swab them with semi-solid IC, AC[arg] vegetable shortening. **Insert** the lubricated tap in hole, **aligning** it carefully. IC[fa:IC] Check the tap against a square to make sure it is straight. IC[pc-to:IC] AT[pc-to:AC[oa]] 2. Use a tap wrench to turn tap clockwise.  $\mathbf{AT}[for]$ *For* first few turns, **exert** moderate downward pressure. After each turn or so, **back** tap *out a bit*. AC[p,adv] File burrs from edge of hole, brush away filings, and add more lubricant to the tap to keep it from breaking off in the hole. AC[pp], AC[adv], AC[arg,pc-to:IC]
- 3. Continue the process of turning the tap, backing it out, brushing metal chips out of the threads, and adding lubricant. AT [oa] If you are threading a blind hole, as you near the bottom, remove tap *completely* after each turn or two, and use a piece of wire or a cotton swab to clean out metal chips. IC[adv], AT[pc-to:IC]

#### Chapter Seven: Concrete and asphalt

#### Mixing by hand

- Using a square ended shovel, spread the premeasured sand evenly on the mixing area, add the required amount of cement, and mix until you get a mass of uniform color without brown or gray streaks.
   AC[arg,pp], AC[arg], AC[until]
   Add coarse aggregate and turn the materials over at least three times or until all the aggregate is evenly distributed.
   AC[arg], AC[p,iter,until]
- Form a shallow depression in the middle of the sand-cement aggregate mixture; then slowly pour *in* some of the measured water and work it *in well*, making *sure* to reach all the way to the bottom of the mound. IC, AC[p,arg], AC[p,adv,fa:IC]
- 3. Pour more water into the depression, pull dry ingredients from the sides of the ring into the water, and mix well. AC[arg], AC[arg], AC[adv]
   Continue adding water, a little at a time, and mixing the materials until they are thoroughly combined and evenly moist. AT[until]
   When all the water has been absorbed, turn the batch three or four times to ensure a uniform mix. AC[iter,pc-to:IC]

#### Chapter Eight: Masonry, building with brick, block, and stone

Mixing and handling mortar

- Cut a slice of mortar from the mound on the mortarboard, using a sawing motion of the trowel. AC[arg]
   With the back of the trowel blade, shape the mortar into a "sausage" about the length and width of the blade. IC
- Load the trowel by sweeping it under the mortar slice from behind with a smooth forward motion. IC[pc-by:AC[pp]]
   As you lift the trowel, snap your wrist down slightly to bond the mortar to the trowel blade. IC[as:AC[arg],adv,pc-to:IC]
- 3. To throw a mortar line, set trowel tip, face up, where line is to begin. IC[pc-to:IC] As you pull trowel toward you, turn blade 180 degrees. AC[as:AC[pp],arg]
- 4. **Furrow** the mortar gently by **running** trowel tip, face down, along center of line. IC[pc-by:AC[pp]]
- 5. To butter a brick before laying it, hold it upright and tilted at a slight angle. AT[pc-to:IC,oa]

**Pick** up a small amount of mortar on the trowel and **swipe** it *onto* the end of the brick. **IC**[p], **AC**[pp]

Squash mortar down *against* all four edges. AC[pp]

6. Shove brick *in* place, its buttered end pressed against adjoining brick.
Continue pressing *until* head and bed joints are the right thickness.
Trim excess mortar with trowel edge; use excess *to* butter next brick.
AT[pc-to:IC]
AC[pp]
AC[pp]
AC[pp]
AT[until]

#### Chapter Ten: Plumbing repairs and installations

#### Unclogging sink drains

1.	<b>Remove</b> stopper or strainer, and <b>block</b> the overflow opening with wet clo	th to create
	a vacuum. IC, I	IC[pc-to:IC]
	<b>Position</b> plunger over drain, and <b>cover</b> cup with water.	IC, IC
	Tilt cup to release trapped air.A	AC[pc-to:IC]
	<b>Plunge</b> forcefully up and down 10 times; <b>remove</b> the plunger abruptly.	$\mathbf{IC}[\mathrm{iter}],\mathbf{IC}$
	Repeat several times.	$\mathbf{AT}[\mathrm{iter}]$
2.	If drain is still plugged, <b>place</b> a bucket under trap, <b>unscrew</b> clean-out p the water <i>drain out</i> . <b>IC</b> , I	lug, and let IC, AC[arg]
	(If trap has no clean-out plug, <b>remove</b> the entire trap, as in step 4.)	IC
	<b>Probe</b> inside trap and pipe with bent wire to free clog.	AT [pc-to:IC]
	Screw plug back in (or reconnect trap).	$\mathbf{AC}[\mathbf{p}],  \mathbf{IC}$
3.	If the problem persists, <b>feed</b> auger through drain hole, <b>cranking</b> the t clockwise, $until$ it hits clog — an area of mushy resistance. <b>AT</b> [fa: A	ool's handle <b>AT</b> [oa],until]
	Work auger back and forth to break $up$ the clog, then flush drain with $AT[pc-to:IC[p]]$ , IC	h hot water.
4.	If drain is still clogged, <b>place</b> a bucket beneath trap to <b>catch</b> water.	IC[pc-to:IC]
	Holding trap in place, use a wrench with taped jaws to unscrew slip nu AT[fa:AT[oa],pc-to:IC]	its.

Remove trap, drain it, and clean it; replace washers if worn.IC, IC, IC, ICFeed auger into pipe in wall and break up blockage.AT[pp,oa], IC[p]Reassemble trap.IC

## B.2 [USAF, 1988]

#### ORGANIZATIONAL MAINTENANCE JOB GUIDE FUEL SYSTEM DISTRIBUTION USAF SERIES F-16C/D AIRCRAFT

## 10 OCTOBER 1988 CHANGE 5 6 SEPTEMBER 1991

#### CONNECTION OF HYDRAULIC TEST STAND

1.	Open access door 3202. IC
2.	$\begin{array}{llllllllllllllllllllllllllllllllllll$
3.	Open access door 3216. IC
4.	<b>Connect</b> hydraulic test stand pressure and return couplings to system A ground test manifold.
5.	Position FFP control valve handle in down (closed) position. IC
6.	Open access door 3101. IC
7.	$\begin{array}{llllllllllllllllllllllllllllllllllll$
8.	Open access door 3115. IC
9.	<b>Connect</b> hydraulic test stand pressure and return couplings to system B ground test manifold.
10.	Connect cooling air. IC
11.	Connect electrical power (paragraph 9). IC

#### REMOVAL OF CROSSFEED VALVE

1.	(A) <b>Remove</b> access panel $3428$ . (General Maintenance)	$\mathbf{IC}$
2.	(A) <b>Purge</b> A1 fuel tank (T.O. 1-1-3).	IC
3.	(A) <b>Disconnect</b> valve fuel tube from valve.	IC
4.	(A) <b>Remove</b> two couplings and <b>slide</b> two sleeves $on$ engine supply tubes.	and lower fuel $IC, AC[pp]$
5.	(A) <b>Note</b> position of any washers; then <b>remove</b> two bolts, aft clamp, a required) from valve support bracket.	and washers (as IC, IC
6.	(A) <b>Loosen</b> two bolts on forward clamp.	IC
7.	(A) Slide valve aft and remove.	$\mathbf{AC}[\text{oa}],\mathbf{IC}$
8.	(A) <b>Remove</b> and <b>discard</b> four packings.	IC, IC

#### **INSTALLATION OF FUEL PUMP NO. 4**

1. (A) Lubricate and install packing (M25988/1-904) on elbow.	IC, IC
2. (A) Install elbow on pump. Do not torque jamnut.	$\mathbf{IC}$
3. (A) <b>Prepare</b> pump and bulkhead mating surfaces for electrical bonding.	$\mathbf{IC}$
4. (A) <b>Connect</b> electrical connector.	$\mathbf{IC}$
5. (A) <b>Position</b> pump on bulkhead.	$\mathbf{IC}$

6.	(A,B) Install four bolts, four washers, four sealing washers, four washers,	and four
	nuts.	IC
	Torque to 50-100 inch-pounds.	$\mathbf{AC}[pp]$
7.	(A) <b>Connect</b> sense tube.	IC
	Torque to 72-78 inch-pounds.	AC[pp]
8.	(A) <b>Torque</b> jamnut to 72-78 inch-pounds.	$\mathbf{AC}[pp]$
9.	(A) Lubricate and install four packings (M25988/1-226), two on connection one on union, and one on pump.	on tube, IC, IC
10.	(A) Lubricate and install two packings (M25988/1-017), one on ejector pum flow tube and one on connection tube.	p motive IC, IC
11.	(A) <b>Position</b> connection tube and two sleeves and <b>install</b> two couplings.	IC, IC
12.	(A) <b>Install</b> clamp, two bolts, and two nuts.	IC
	Torque to 40-60 inch-pounds.	$\mathbf{AC}[\mathrm{pp}]$
13.	(A) <b>Position</b> sleeve and <b>install</b> coupling.	IC, IC
14.	(A) Lubricate and install four packings (M25988/1-017), two on turbine pum flow tube, one on connection tube, and one on turbine pump fuel fitting.	p motive IC, IC
15.	(A) <b>Position</b> turbine pump motive flow tube and two sleeves and <b>install</b> two co <b>IC</b> , <b>IC</b>	ouplings.
16.	(A) <b>Position</b> clamp and <b>install</b> bolt.	IC, IC
	Torque to 40-60 inch-pounds.	$\mathbf{AC}[\mathrm{pp}]$
17.	(A) Install access panel 3426 using 45 bolts. (General Maintenance)	IC
18.	(A) Install access panel 3428 using 45 bolts. (General Maintenance)	IC
19.	(A) <b>Perform</b> fuel tank access panel leak check. (General Maintenance)	IC
CHECKC	OUT OF FUEL FLOW PROPORTIONER PRESSURE SWITCH	
1.	(B) <b>Connect</b> hydraulic test stand to system A. (General Maintenance)	IC
2.	(B) <b>Position</b> FFP control valve handle (access door 3216) to up (open) posit	ion. IC
3.	(A) <b>Position</b> main power switch to MAIN PWR.	IC
4.	(A,B) Increase hydraulic pressure $to 3000$ psig as indicated on HYD PRESS cator.	3 A indi- AC[pp]
5.	(A) <b>Position</b> ENG FEED switch to NORM.	IC
6.	(A) <b>Inspect</b> pressure switch.	IC
7.	(A) <b>Position</b> ENG FEED switch to OFF.	IC
8.	(A) <b>Position</b> main power switch to OFF.	IC
9.	(B) <b>Disconnect</b> hydraulic test stand. (General Maintenance)	IC
REMOVA	AL OF WING FUEL PUMPS	

1. (A) <b>Remove</b> access cover 5419 (left) or 6420 (right). (General M	aintenance) <b>IC</b>
2. (A) <b>Raise</b> handle on cartridge and $\mathbf{turn}$ to unlock position.	$\mathbf{AC}[\mathrm{oa}],  \mathbf{AC}[\mathrm{pp}]$
3. (A) <b>Pull</b> straight up on handle <i>until</i> cartridge is clear of housing.	$\mathbf{AT}[\mathrm{until}]$
4. (A) <b>Remove</b> and <b>discard</b> four packings.	IC, IC

## B.3 [ITL SIMA, 1997]

## Mitre saw assembly line instructions

## http://speckle.ncsl.nist.gov/~sima/vim/

## Build the table assembly.

 $\mathbf{Put}$ 

1.	Press the button <i>onto</i> the spindle lock.	$\mathbf{C}[pp]$
2.	Slip an O-ring onto the locking pin.A	$\mathbf{C}[pp]$
3.	<b>Put</b> the tip cover on the detent spring.	$\mathbf{IC}$
4.	<b>Put</b> a table on the fixture <i>such that</i> the bottom faces up.	$\mathbf{C}[ms]$
5.	Pound the miter pointer on with a hammer.	$\mathbf{C}[pp]$
6.	<b>Align</b> the 2 wear plates along the edges and <b>screw</b> them <i>down</i> with 4 screws. $\mathbf{AC}[p]$	$\mathbf{IC},$
7.	Insert the knob, put the clamp plate on just below it, and drive the 2 screws. IC, AC[oa]	$\mathbf{IC},$
8.	Set the detent spring in place, screw it $in$ with 2 screws, and flip the fixture. AC[p], IC	$\mathbf{IC},$
9.	Attach the bevel pointer and bracket with a screw.	$\mathbf{IC}$
10.	Near the trunnion end, $\mathbf{drive}$ the 2 adjusting screws $to$ the nut.	$\mathbf{C}[pp]$
11.	Flip the fixture to vertical and grease the trunnion's place. IC, A	$\mathbf{C}[\mathrm{arg}]$
12.	<b>Apply</b> Loctite to the stud and <b>drive</b> it into the end of the table. $AC[pp], A$	$\mathbf{C}[pp]$
13.	Put the trunnion in position and screw it <i>into</i> place. IC, A	$\mathbf{C}[pp]$
	(a) <b>Make sure</b> that the trunnion can easily rotate.	$\mathbf{IC}$
14.	<b>Place</b> the assembly on the next line on the belt conveyor.	IC
the a	assembly onto the base.	
1.	Get a base and grease its hole with a brush. IC, A	$\mathbf{C}[\mathrm{arg}]$
2.	With the rounded part of the base toward you, <b>push</b> the curved spring <i>in</i> it $\mathbf{AC}[\mathbf{p}]$	s slot.
3.	Set the table assembly on top of the base <i>such that</i> the knob faces you.	$\mathbf{IC}[ms]$
4.	Get a dial indicator.	$\mathbf{IC}$
	(a) Occasionally <b>calibrate</b> the dial indicator on the flat surface provided.	$\mathbf{IC}$
5.	Place the detent plate and drive the 3 screws into it just a little bit. IC, A	$\mathbf{C}[pp]$
6.	Rotate the table $to$ the "0" mark on the detent plate and tighten the knob. A IC	<b>C</b> [pp],
7.	Put the dial indicator on the table and level the trunnion.	$\mathbf{C}, \mathbf{IC}$
8.	Place the base fence and drive the 4 screws. IC, A	AC[oa]
	(a) <b>Be sure</b> to drive the inner 2 screws first <i>to</i> <b>prevent</b> warping. $IC$ [pc-to: <b>A</b>	$\mathbf{C}[oa]]$
9.	<b>Place</b> the left fence in its slot.	$\mathbf{IC}$
10.	<b>Put</b> the lower bevel knob on the fence.	$\mathbf{IC}$

12.	<b>Tighten</b> both knobs with a screw gun.	$\mathbf{IC}$
	(a) With the dial indicator, <b>make sure</b> that the fence is perpendicular to the <b>IC</b>	able.
	(b) <b>Check</b> the table's tightness.	$\mathbf{IC}$
13.	According to your shift, <b>mark</b> the unit with a marker.	$\mathbf{IC}$
14.	Affix the left and right warning labels to the detent plate.	$\mathbf{IC}$

# Appendix C PAR translation for SPUD

Here is pseudo-code, written loosely in C notation, for translating the PAR feature structure representation into modal first-order logic for SPUD. Only the **result** attribute is considered common knowledge (thus using the C modal operator); all other action information is considered private to the system (the S modal operator). A few simple functions are assumed to be defined for reading in the feature structure notation:

- getVal(string) returns the value for the attribute named by string in the top-level feature structure. The returned value could be a feature structure, a list, an atom (such as a string or a number), or the NULL object if the attribute is not found.
- getValInFS(FS, string) returns the value for the attribute named by string in the feature structure named by FS. The NULL object is returned if the attribute is not found.
- getValInList(list, string) returns the value for the item in list identified by string. The NULL object is returned if the identifier is not found.

nonEmpty(var) returns false if var is the NULL object and true otherwise.

The parts of PAR which are not used at all in this implementation are ignored, but could be translated in the same fashion.

```
## START
ACTID = getVal("id");
if (!nonEmpty(ACTID)) ACTID = assignID();
DTID = getVal("during");
if (nonEmpty(DTID)) printf("S during(%s, %s).\n", ACTID, DTID);
RTID = getVal("result");
if (nonEmpty(RTID)) {
  printf("C result(%s, %s).\n", ACTID, RTID);
  printf("S meets(%s, %s).\n", RTID, DTID);
}
PART = getVal("participants");
if (nonEmpty(PART)) {
  AGID = getValInFS(PART, "agent");
  if (nonEmpty(AGID))printf("S agent(%s, %s).\n", ACTID, AGID);
  OBJFS = getValInFS("objects");
  if (nonEmpty(OBJFS)) {
    IID = getLabelledValInFS(OBJFS, "instrument");
    if (nonEmpty(IID)) printf("S instrument(%s, %s).\n", ACTID, IID);
  }
}
CS = getVal("core semantics");
if (nonEmpty(CS)) {
  PREC = getValInfs(CS, "precondition");
  if (nonEmpty(PREC)) printf("S postcondition(%s, %s).\n", ACTID, PREC);
  POSTC = getValInfs(CS, "postcondition");
  if (nonEmpty(POSTC)) printf("S postcondition(%s, %s).\n", ACTID, POSTC);
  MOTION = getValInFS(CS, "motion");
  if (nonEmpty(MOTION)) {
    MOBJ = getValInFS(MOTION, "object");
    MTYPE = getValInFS(MOTION, "type");
    printf("S motion(%s, %s, %s).\n", ACTID, MOBJ, MTYPE);
  }
  FORCE = getValInFS(CS, "force");
  if (nonEmpty(FORCE)) {
    FOBJ = getValInFS(FORCE, "object");
    printf("S force(%s, %s).\n", ACTID, FOBJ);
    FMAG = getValInFS(FORCE, "magnitude");
    printf("S magnitude(%s, %s).\n", ACTID, FMAG);
  }
}
```

```
SA = getVal("subactions");
if (nonEmpty(SA)) {
  SAID = getValInFS(SA, "head");
  if (nonEmpty(SAID)) printf("S subactions(%s, %s).\n", ACTID, SAID);
}
NAID = getVal("next action");
if (nonEmpty(NAID)) printf("S nextAction(%s, %s).\n", ACTID, NAID);
PAID = getVal("previous action");
if (nonEmpty(PAID)) printf("S prevAction(%s, %s).\n", ACTID, PAID);
PARENTID = getVal("parent action");
if (nonEmpty(PAID)) printf("S parent(%s, %s).\n", ACTID, PAID);
if (definedAction(SAID)) insertTimeReln("starts", ACTID, SAID).
if (!nonEmpty(NAID)) insertTimeReln("finishes", ACTID, PARENTID).
```

## END

# Index

action representation, 55–63 improving, 141 independently-motivated, 3 PAR, 10, 56-59 PAR translation, 59 step predicate, 96 substep predicate, 97 time intervals, 60–61 actions accomplishments, 15 achievements, 15 activities, 15 classes, 5 concreteness, 62–63 culmination, 5, 17 descriptions, 5 instances, 5 interactions and termination, 40 interactions between, 8 termination, 61–62 adverbs, 38 as source of termination, 43 agent expertise, 63-71 experience levels, 64–65

communicative goals, 31 action relations, 104 and effective instructions, 125 and the SPUD algorithm, 93 implementation issues, 126 unique identification, 91–93 computation time, 123 conjunction of sentences, see discourse segments, with multiple sentences of verb phrases, 90 corpus analysis, 26 culmination, see action, culmination acquired, 37 inherent, 36 description logics, 19 discourse segments for emphasizing sub-actions, 87 implementation of, 78-79 with multiple sentences, 90–91 distractors, 2, 91 domain representation, 32 domain statements, 91 dynamic semantics, 19

effective instructions, 33 and communicative goals, 125 seeinstructions, effective, 1 evaluation criteria, see implementation, evaluation criteria events, 16 culmination, see actions, culmination tripartite structure, 17 feature structures, 19 first-order logic (FOL), 20 free adjuncts, 39 as source of termination, 43 generation instances, 95–96 generation process example, 101 implementation evaluation criteria, 142 scalability, 143 implementation issues, 125–126 algorithm modifications, 126 communicative goals, 126 designing trees, 125 stylistic preferences, 125 instructions, 5 effective, 1 instruction step, 5, 35 step-by-step, 6, 35 language-neutrality, 32

lexical choice, 21–23 lexical items implementation of, 74–76 multi-clausal, 84 linguistic realization, 32 LTAG, 26 adjunction, 27 alpha trees, 27 beta trees, 27 example of tree operations, 28 means clauses, 45 implementation of, 86 modal logic, 20 modal operators, 21 morphological processing, 31 noun phrases for empty subject, 81 implementation of, 81–84 object representation, 52–55 objects abstract, 55 physical, 52 properties, 53 reasoning about, 54 PAR, see action representation, PAR predicate-argument structure, 7 prepositional phrases, 38 implementation of, 81

purpose clauses coding, 39 fronted, 24, 46, 87-88 implementation of, 86-88 means clauses, seemeans clauses86 non-fronted, 24, 46, 86-87 referring expressions, 2 scalability, see implementation, scalability sentence structure implementation of, 79 sources of termination analysis, 43-49 SPUD, 11 algorithm, 91-93 possible modifications, 126, 144 stylistic preferences, see also views of actions implementation issues, 125 syntactic constructions implementation of, 76–78 TAG, 26 temporal relations, 17 finishes, 17 meets, 17 starts, 17 termination

acquired, 37

termination conditions, 4, 47 termination information, 2 time intervals, see action representation, time intervals until clauses, 40, 47 implementation of, 88 verb arguments, 38 as source of termination, 43 verb particles, 38 as source of termination, 44 verbs implementation of, 79–81 views of actions, 84-85 complex, 85 elaborate, 85 purpose, 85

## Epilogue

Paraphrasing from the end of Marc Moens' dissertation [Moens, 1987], writing a dissertation seems to be an activity which does not culminate; at some point, it's just over.

## Colophon

This dissertation was typeset using  $L^{AT}EX(2e)$  with the following style files or packages:

- the standard **report** style file,
- the **11pt** style file,
- a version of the **penndiss** style file (mostly recently modified by I. Dan Melamed and Jeff Reynar in April 1998), used to conform to Penn's dissertation requirements,
- the **lingmacros** style file (modified 4/1/91 by Emma Pease), used for typesetting example sentences,
- the **epsf** style file for including encapsulated PostScript files,
- the **covingtn** style file (Michael A. Covington's linguistic macros, November 1992), used for typesetting feature structures,
- the **QobiTree** package (written by Jeffrey M. Siskind), used for typesetting syntactic trees,
- the **aaai-named** style file for typesetting citations and the bibliography, and
- the **makeidx** style file and **makeindex** Unix program, used for making the index.