Chapter 6

Task-Level Specifications

So far we have been talking about real-time interactive display and manipulation of human figures, with the goal of enabling human factors engineers to augment their analyses of designed environments by having human figures carry out tasks intended for those environments. This chapter explores the use of task-level specifications as an alternative to direct manipulation for generating task simulations.

By now, the reader should be convinced of the value of being able to simulate, observe and evaluate agents carrying out tasks. The question is what is added by being able to produce such simulations from high-level task specifications. The answer is efficient use of the designer's expertise and time. A designer views tasks primarily in terms of what needs to be accomplished, not in terms of moving objects or the agent's articulators in ways that will eventually produce an instance of that behavior -- e.g., in terms of slowing down and making a left turn rather than in terms of attaching the agent's right hand to the clutch, moving the clutch forward, reattaching the agent's right hand to the steering wheel, then rotating the wheel to the left and then back an equal distance to the right. As was the case in moving programming from machine-code to high-level programming languages, it can be more efficient to leave it to some computer system to convert a designer's high-level goal-oriented view of a task into the agent behavior needed to accomplish it. Moreover, if that same computer system is flexible enough to produce agent behavior that is appropriate to the agent's size and strength and to the particulars of any given environment that the designer wants to test out, then the designer is freed from all other concerns than those of imagining and specifying the environments and agent characteristics that should be tested.

This chapter then will describe a progression of recent collaborative efforts between the University of Pennsylvania's Computer Graphics Research Lab and the LINC Lab (Language, Information and Computation) to move towards true high-level task specifications embodying the communicative richness and efficiency of Natural Language instructions.

The first section will describe our earliest work on using simple Natural
Language commands to drive task animation. This work also dealt with an aspect of performance simulation that is rarely communicated in task specifications: how long an action will take an agent to perform. Section 6.2 describes our next effort at using Natural Language commands to drive task animation, focusing on how kinematic and spatial aspects of desired behavior are conveyed in Natural Language commands.

One of the consequences of these first two studies is understanding the value of a stratified approach to mapping from language to behavior: it is not efficient for, say, the language understanding components to make decisions that commit the agent moving a hand or a knee in a particular way, unless those movements are stated explicitly (but rarely) in the text. Because of this recognized need for intermediate representations between Natural Language descriptions and animation directives, an experiment was performed, described in Section 6.3, in which an intermediate level compositional language was created for specifying task-actions and generating task-level animated simulations from scripts written in this language. This demonstration paves the way for the ultimate connection between the behavioral simulation structure of the preceding Chapter and the conceptual structures of this one.

Each of these early efforts focused on individual commands in Natural Language. Task specifications, on the order of Natural Language instructions, go beyond individual commands in specifying what an agent should (and shouldn’t) do. Since our current work is aimed at generating task animations from specifications as rich as Natural Language instructions, we devote the discussion in Section 6.4 to describing some features of instructions and what an understanding system requires in order to derive from them what a person would in understanding and acting in response to instructions.

6.1 Performing Simple Commands

Our first work on driving task animation through Natural Language commands was a prototype system developed by Jeffrey Esakov that explored simple relations between language and behavior [EBJ89]. In this case, the desired behaviors were simple arm reaches and head orientation (view) changes on the part of the animated figures. While seemingly very easy, these tasks already demonstrate much of the essential complexity underlying language-based animation control.

6.1.1 Task Environment

In Esakov’s work, the tasks to be animated center around a control panel (i.e. a finite region of more or less rigidly fixed manually-controllable objects) – here, the remote manipulator system control panel in the space shuttle with its variety of controls and indicators. Because Esakov was producing task animations for task performance analysis, he needed to allow performance

\footnote{Jeffrey Esakov.}
to depend upon the anthropometry of the agent executing the task. In the experiments, all the controls were in fact reachable without torso motion by the agents being animated; failure situations were not investigated and the fully articulated torso was not yet available. An animation of one of the experiments can be found in [EB91].

6.1.2 Linking Language and Motion Generation

The primary focus of this work was to combine Natural Language task specification and animation in an application-independent manner. This approach used the following Natural Language script:

John is a 50 percentile male.
Jane is a 50 percentile female.
John, look at switch twf-1.
John, turn twf-1 to state 4.
Jane, look at twf-3.
Jane, look at tglJ-1.
Jane, turn tglJ-1 on.
John, look at tglJ-2.
Jane, look at twf-2.
Jane, turn twf-2 to state 1.
John, look at twf-2.
John, look at Jane.
Jane, look at John.

(The abbreviations denote thumbwheels such as twf-1 and toggle switches such as tglJ-1. Thumbwheels have states set by rotation; toggles typically have two states, on or off.)

This type of script, containing simple independent commands, is common to checklist procedures such as those done in airplanes or space shuttles [Cen81]. The verb "look at" requires a view change on the part of the figure, and the verb "turn" requires a simple reach. (Fine hand motions, such as finger and wrist rotations, were not animated as part of this work.) The two primary problems then are specifying reach and view goals, and connecting object references to their geometric instances.

6.1.3 Specifying Goals

For these reach tasks the goal is the 3D point which the fingertips of the hand should touch. A view goal is a point in space toward which one axis of an object must be aimed. Spatially, such goals are just Peabody sites and must be specified numerically with respect to a coordinate system. Within the natural language environment, goals are not seen as coordinates, but rather as the objects located there – for example,

John, look at switch twf-1.
Jane, turn switch tglJ-1 on.
Because the exact locations of the switches is unimportant at the language level, in creating an animation, the switch name tglJ-1 must be mapped to the appropriate switch on the panel in the animation environment. The same process must be followed for the target object toward which an object axis must be aligned in a view change. This problem reduces to one of object referencing.

6.1.4 The Knowledge Base

In general, all objects have names. Since the names in the task specification environment may be different from those in the animation environment, there must be a mapping between the names. The knowledge base that Esakov used contained information about object names and hierarchies, but not actual geometry or location. He used a frame-like knowledge base called DC-RL to store symbolic information [Ceb87]. For example, the DC-RL code for an isolated toggle switch, tglJ-1, follows:

```prolog
{ concept tglJ-1 from control
  having ( 
    [role name with [value = "TOGGLE J-1"]]
    [role location with [value = panel1]]
    [role type-of with [value = switch]]
    [role sub-type with [value = tgl]]
    [role direction with [value = (down up)]]
    [role states with [value = (off on)]]
    [role movement with [value =
       (discrete mm linear ((off on) 20 5))]]
    [role current with [value = off]]
  )
}
```

To reference this switch from within the animation environment, a mapping file was generated at the same time the graphical object was described.

```prolog
{ concept ctrlpanel from panelfig
  having ( 
    [role twF-1 with
      [ value = ctrlpanel.panel.twf_1 ]]
    [role twF-2 with
      [ value = ctrlpanel.panel.twf_2 ]]
    [role twF-3 with
      [ value = ctrlpanel.panel.twf_3 ]]
    [role tglJ-1 with
      [ value = ctrlpanel.panel.tglj_1 ]]
    [role tglJ-2 with
      [ value = ctrlpanel.panel.tglj_2 ]]
  )
}
```
6.1. PERFORMING SIMPLE COMMANDS

The names twf-1, twf-2, tglj-1 correspond to the names of switches in the existing knowledge base panel description called panel.fig. These names are mapped to the corresponding names in the animation environment (e.g., ctrlpanel.panel.twf-1, etc.) and are guaranteed to match.

6.1.5 The Geometric Database

The underlying geometric database is just Peabody code. The salient toggle and thumbwheel locations were simply mapped to appropriate sites on a host segment representing the control panel object. The relevant part of the Peabody description of the panel figure is shown:

```text
figure ctrlpanel {
  segment panel {
    psurf = "panel.pss";
    site base->location =
      trans(0.00cm,0.00cm,0.00cm);
    site twf_1->location =
      trans(13.25cm,163.02cm,80.86cm);
    site twf_2->location =
      trans(64.78cm,115.87cm,95.00cm);
    site twf_3->location =
      trans(52.84cm,129.09cm,91.43cm);
    site tglj_1->location =
      trans(72.36cm,158.77cm,81.46cm);
    site tglj_2->location =
      trans(9.15cm,115.93cm,94.98cm);
  }
}
```

This entire file is automatically generated by a modified paint program. Using the panel as a texture map, switch locations are interactively selected and the corresponding texture map coordinates are computed as the site transformation. The panel itself is rendered as a texture map over a simple polygon and the individual sites then refer to the appropriate visual features of the switches.

6.1.6 Creating an Animation

Linking the task level description to the animation requires linking both object references and actions. A table maps the names of objects from the task description environment into the psurf geometry of the animation environment. In this simple problem domain the language processor provides the other link by associating a single key pose with a single animation command. Each part of speech fills in slots in an animation command template. Simple Jack behaviors compute the final posture required by each command which are then strung together via simple joint angle interpolation.
6.1.7 Default Timing Constructs

Even though the basic key poses can be generated based upon a Natural Language task description, creating the overall animation can still be difficult. We have already discussed posture planning and collision avoidance issues, but there is yet another problem that bears comment. From the given command input, the timing of the key poses is either unknown, unspecified, or arbitrary.

Action timings could be explicitly specified in the input, but (language-based) task descriptions do not normally indicate time. Alternatively, defining the time at which actions occur can be arbitrarily decided and iterated until a reasonable task animation can be produced. In fact, much animator effort is normally required to temporally position key postures. There are, however, more computational ways of formulating a reasonable guess for possible task duration.

Several factors effect task performance times, for example: level of expertise, desire to perform the task, degree of fatigue (mental and physical), distance to be moved, and target size. Realistically speaking, all of these need to be considered in the model, yet some are difficult to quantify. Obviously, the farther the distance to be moved, the longer a task should take. Furthermore, it is intuitively accepted that performing a task which requires precision work should take longer than one not involving precision work: for example, threading a needle versus putting papers on a desk.

Fitts [Fit54] and Fitts and Peterson [FP64] investigated performance time with respect to two of the above factors, distance to be moved and target size. It was found that amplitude ($A$, distance to be moved) and target width ($W$) are related to time in a simple equation:

\[
\text{Movement Time} = a + b \log \frac{2A}{W} \quad (6.1)
\]

where $a$ and $b$ are task-dependent constants. In this formulation, an index of movement difficulty is manipulated by the ratio of target width to amplitude and is given by:

\[
\text{ID} = \log \frac{2A}{W} \quad (6.2)
\]

This index of difficulty shows the speed and accuracy tradeoff in movement. Since $A$ is constant for any particular task, to decrease the performance time the only other variable in the equation $W$ must be increased. That is, the faster a task is to be performed, the larger the target area and hence the movements are less accurate.

This equation (known as Fitts' Law) can be embedded in the animation system, since for any given reach task, both $A$ and $W$ are known. The constants $a$ and $b$ are linked to the other factors such training, desire, fatigue, and body segments to be moved; they must be determined empirically. For button tapping tasks, Fitts [FP64] determined the movement time ($MT$) to be

\[
MT_{\text{arm}} = 74ID - 70\text{msec} \quad (6.3)
\]
6.1. PERFORMING SIMPLE COMMANDS

<table>
<thead>
<tr>
<th>Actor</th>
<th>Action</th>
<th>ID</th>
<th>Fitts Duration</th>
<th>Scaled Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>Look twf-1</td>
<td>2.96</td>
<td>321</td>
<td>963</td>
</tr>
<tr>
<td>John</td>
<td>Turn twf-1</td>
<td>5.47</td>
<td>335</td>
<td>1004</td>
</tr>
<tr>
<td>John</td>
<td>Look tglJ-2</td>
<td>4.19</td>
<td>566</td>
<td>1697</td>
</tr>
<tr>
<td>John</td>
<td>Look twf-2</td>
<td>4.01</td>
<td>530</td>
<td>1500</td>
</tr>
<tr>
<td>John</td>
<td>Look Jane</td>
<td>4.64</td>
<td>655</td>
<td>1968</td>
</tr>
<tr>
<td>Jane</td>
<td>Look twf-3</td>
<td>4.28</td>
<td>584</td>
<td>876</td>
</tr>
<tr>
<td>Jane</td>
<td>Look tglJ-1</td>
<td>3.64</td>
<td>456</td>
<td>685</td>
</tr>
<tr>
<td>Jane</td>
<td>Turn tglJ-1</td>
<td>5.39</td>
<td>329</td>
<td>493</td>
</tr>
<tr>
<td>Jane</td>
<td>Look twf-2</td>
<td>4.16</td>
<td>560</td>
<td>840</td>
</tr>
<tr>
<td>Jane</td>
<td>Turn twf-2</td>
<td>4.99</td>
<td>299</td>
<td>449</td>
</tr>
<tr>
<td>Jane</td>
<td>Look John</td>
<td>4.33</td>
<td>594</td>
<td>891</td>
</tr>
</tbody>
</table>

Table 6.1: Task Durations Using Fitts' Law.

In determining this equation, it was necessary to filter out the extraneous factors. This was done by having the subjects press the button as quickly as possible and allowing them to control the amount of time between trials. Jagacinski and Monk [JM85] performed a similar experiment to determine the movement time for the head and obtained the following equation:

\[
\begin{align*}
MT_{\text{head}} &= 109ID' - 268\text{ms} \\
ID' &= \log \frac{2A}{W-W_0} 
\end{align*}
\]  

(6.4)  

(6.5)

This equation is the result of equating the task to inserting a peg of diameter \( W_0 \) into a hole of diameter \( W \), and resulted in a better fit of the data.

For our purposes the above constants may not apply. Since it was our desire to have the man in our animation move sluggishly and the woman move quickly (but not too quickly), we scaled Equations 6.3 and 6.4 by differing constants:

\[
\begin{align*}
MT_{\text{man(arm)}} &= 3 \cdot MT_{\text{arm}} \\
MT_{\text{man(head)}} &= 3 \cdot MT_{\text{head}} \\
MT_{\text{woman(arm)}} &= 1.5 \cdot MT_{\text{arm}} \\
MT_{\text{woman(head)}} &= 1.5 \cdot MT_{\text{head}}
\end{align*}
\]

This width of the target, \( W \) in equation 6.2 was chosen to be 1cm. For head movements, we chose \( W_0 = .33^\circ \) after [JM85]. This results in the action durations shown in Table 6.1.

Although Fitts' Law has been found to be true for a variety of movements including arm movements (\( A = 5 - 30\) cm), wrist movements (\( A = 1.3\) cm)
[Dru75, JM85, LCF76], and head movements \((A = 2.45 - 7.50^\circ)\) [JM85] the application to 3D computer animation is only approximate. The constants differ for each limb and are only valid within a certain movement amplitude in 2D space, therefore the extrapolation of the data outside that range and into 3D space has no validated experimental basis. Nonetheless, Fitts’ Law provides a reasonable and easily computed basis for approximating movement durations.

While it may seem odd at first to have attacked both Natural Language interpretation and timing constructs as part of the same research, Esakov’s work foreshadows our more recent work on language and animation by focusing on the fact that the same instruction, given to agents with different abilities, will be carried out in different ways. Language tells an agent what he or she should attempt to do: how he or she does it depends on them.

### 6.2 Language Terms for Motion and Space

The next piece of work that was done on driving task animation through Natural Language commands was Jugal Kalita’s work on how Natural Language conveys desired motion and spatial aspects of an agent’s behavior. In English, the primary burden falls on verbs and their modifiers. Kalita’s work showed how verbs and their modifiers can be seen as conveying spatial and kinematic constraints on behavior, thereby enabling a computer to create an animated simulation of a task specified in a Natural Language utterance. This work is described at greater length in [Kal90, KB90, KB91].

#### 6.2.1 Simple Commands

To understand Kalita’s contribution, consider the following commands:

- *Put the block on the table.*
- *Turn the switch to position 6.*
- *Roll the ball across the table.*

Each of these commands specifies a task requested of an agent. Performing the task, requires *inter alia* that the agent understand and integrate the meanings of the verbs (*put, turn, open, roll*) and the prepositions (*on, to, across*). This requires understanding the significant geometric, kinematic or dynamic features of the actions they denote.

In Kalita’ approach to physically based semantics, a motion verb denotes what may be called a *core* or *kernel* action(s). This kernel representation is then used with object knowledge and general knowledge about actions to obtain semantic representations and subsequent task performance plans which are specific to a context – for example,

---

2 Jugal Kalita.
6.2. LANGUAGE TERMS FOR MOTION AND SPACE

- The core meaning of the verb *put* (as in *Put the block on the table*) establishes a geometric constraint that the first object (here, the block) remains geometrically constrained to (or, brought in contact with and supported by) a desired position on the second object (here, the table).

- The core meaning of the verb *push* (as in *Push the block against the cone*) involves applying a force on the manipulated object (here, the block) through the instrument object (here, the hand). The prepositional phrase specifies the destination of the intended motion.

- The verb *roll* (as in *Roll the ball across the table*) involves two related motions occurring simultaneously—one rotational about some axis of the object, and the other translational, caused by the first motion. The rotational motion is repeated an arbitrary number of times.

6.2.2 Representational Formalism

Geometric relations and geometric constraints

The meanings of locative prepositions are represented using a template called a *geometric-relation*. A simple geometric relation is a frame-slot structure:

```
geometric-relation: spatial-type:
  source-constraint-space:
  destination-constraint-space:
  selectional-restrictions:
```

*Spatial-type* refers to the type of the geometric relation specified. Its values may be *positional* or *orientational*. The two slots called *source-constraint-space* and *destination-constraint-space* refer to one or more objects, or parts or features thereof, which need to be related. For example, the command *Put the cup on the table* requires one to bring the bottom surface of the cup into contact with the top surface of the table. The command *Put the ball on the table* requires bringing an arbitrary point on the surface of the ball in contact with the surface of the table top. Since the items being related may be arbitrary geometric entities (i.e., points, surfaces, volumes, etc.), we call them *spaces*. The first space is called the *source-constraint space* and the second, the *destination-constraint space*. The slot *selectional-restrictions* refers to conditions (static, dynamic, global or object-specific) that need to be satisfied before the constraint can be executed.

More complex geometric relations require two or more geometric relations to be satisfied simultaneously:

```
geometric-relation:
  { g-union
    g-relation-1
    g-relation-2
    ...
    g-relation-n
  }
```
where $g$-relation-$i$ is simple or complex.

Geometric relations are also used in the specification of geometric constraints, which are geometric goals to be satisfied:

Geometric-constraint: execution-type:
   geometric-relation:

Geometric constraints are distinguished by their execution-type slot, which can take one of four values: achieve, break, maintain or modify.

Kinematics

The frame used for specifying kinematic aspects of motion is the following:

Kinematics: motion-type:
   source:
   destination:
   path-geometry:
   velocity:
   axis:

Motions are mainly of two types: translational and rotational. In order to describe a translational motion, we need to specify the source of the motion, its destination, the trajectory of its path (path-geometry), and the velocity of the motion. In the case of rotational motion, path-geometry is circular, and velocity, if specified, is angular. Rotational motion requires an axis of rotation. If not specified, it is inferred by consulting geometric knowledge about the object concerned.

Kernel actions

The central part of an action consists of one or more components: dynamics, kinematics and geometric-constraints—along with control structures stating its other features. The control structures used in the examples that follow are: repeat-arbitrary-times and concurrent. The keyword concurrent is specified when two or more components, be they kinematic, dynamic or geometric constraints, need to be satisfied or achieved at the same time. The keyword repeat-arbitrary-times provides a means for specifying the frequentation property of certain verbs where certain sub-action(s) are repeated several times. The verbs' semantic representation need not specify how many times the action or sub-action may need to be repeated. However, since every action is presumed to end, the number of repetitions of an action will have to be computed from simulation (based on tests for some suitable termination conditions), or by inference unless specified linguistically as in Shake the block about fifty times.
6.2. LANGUAGE TERMS FOR MOTION AND SPACE

6.2.3 Sample Verb and Preposition Specifications

Many of the features of Kalita’s representation formalism can be seen in his representation of the verbs “roll” and “open” and the prepositions “in” and “across”. Others can be seen in the worked example in Section 6.2.4.

A kinematic verb: roll

The verb roll refers to two motions occurring concurrently: a rotational motion about the longitudinal axis of the object and a translational motion of the object along an arbitrary path. The rotational motion is repeated an arbitrary number of times. The verb roll is thus specified as:

```
roll (l-agent, l-object, path-relation) ←
  agent: l-agent
  object: l-object
  kernel-action:
    concurrent {{
      kinematic:
        motion-type: rotational
        axis: longitudinal-axis-of (l-object)
    } repeat-arbitrary-times }
    { kinematic:
      motion-type: translational
      path: path-relation }

  selectional restrictions: has-circular-contour (l-object,
                           longitudinal-axis-of (l-object))
```

A verb that removes constraints: open

One sense of open is to move (as a door) from closed position. The meaning is defined with respect to a specific position of a specific object. The closed position of the object can be viewed as a constraint on its position or orientation. Thus, this sense of open involves an underlying action that undoes an existing constraint. The object under consideration is required to have at least two parts: a solid 2D part (the cover) and an unfilled 2D part defined by some kind of frame (the hole). The meaning must capture two things: (1) that at the start of the action, the object’s cover must occupy the total space available in object’s hole in the constrained position, and (2) that the result of the action is to remove the constraint that object’s cover and its hole are in one coincident plane. This is fulfilled by requiring that the two sub-objects (the hole and the cover) are of the same shape and size.

The definition for open is:

```
open (Ag, Obj) ←
  agent: Ag
  object: Obj
  kernel-action:
    geometric-constraint:
```
CHAPTER 6. TASK-LEVEL SPECIFICATIONS

execution-type: break
spatial-type: positional
geometric-relation: source-constraint-space: Obj • hole
destination-constraint-space: Obj • cover
selectional-restrictions: contains-part (Obj, hole)
contains-part (Obj, cover)
area-of (Obj • cover) = area-of (Obj • hole)
shape-of (Obj • cover) = shape-of (Obj • hole)

A locative preposition: in

The sense of in captured here is within the bounds of, contained in or included within. According to Herskovits [Her86], this use type for in is spatial entity in a container. This meaning of in is specified as

in (X,Y) ←
  geometric-relation: positional
  spatial-type: 
  source-constraint-space: volume-of (X)
destination-constraint-space: interior-of (Y)
selectional-restrictions:
or (container-p (Y), container-p (any-of (sub-parts-of (Y))))
size-of (X) ≤ size-of (Y)
normally-oriented (Y)

A container is an object which can hold one or more objects such that the object is "surrounded by" the volume defined by the boundaries of the container. It is a concept which is difficult to define clearly, although heuristics can be devised to recognize whether an object is a container. For our purposes, if an object or any of its part(s) can work as container(s), it will be so labeled in the function slot of its representation. The second selectional restriction is due to Cooper [Coo68]. The third restriction is due to Herskovits, who explains its necessity by stating that the sentence The bread is in the bowl is pragmatically unacceptable if the bowl is upside down and covers the bread under it [Her86].

A path preposition: across

Path is a part of kinematic specification of a motion or an action. A complete definition of path requires specifying its source, destination and path geometry, which Kalita does, using a structure called a path-specification:

path-specification:
  source:
  destination:
  path-geometry:
6.2. LANGUAGE TERMS FOR MOTION AND SPACE

Across is one of several path prepositions in English. Others include from, to, around, round and along. Across has two types of meanings—dynamic and static (locative) meaning. The dynamic meaning implies a journey across an object, whereas the static meaning implies a location between two lines (edges) perpendicular to them and touching, and (possibly) extending beyond them. The dynamic sense of across is seen in:

- Roll/Slide/Move the block/ball across the board.

This dynamic sense of across specifies all three components required for path specification.

\[
\text{across } (X, Y) \leftarrow \text{path-specification}:
\]

\[
source: \text{any-of } (\text{exterior-edges-of } (Y, \text{parallel-to } (\text{longitudinal-axis } (Y))))
\]

\[
destination: \text{any-of } (\text{exterior-edges-of } (Y, \text{parallel-to } (\text{longitudinal-axis } (Y))))
\]

\[
\text{path-geometry: straight-line}
\]

\[
\text{selectional-restrictions:}
\]

\[
\text{destination} \neq \text{source}
\]

\[
\text{has-axis } (X, \text{longitudinal})
\]

\[
\text{angle-between } (\text{path-geometry, longitudinal-axis } (Y), 90^\circ)
\]

\[
\text{length } (Y) \geq \text{width } (Y)
\]

\[
\text{length } (Y) > \text{dimension-of } (X, \text{along-direction } (\text{longitudinal-axis } (Y)))
\]

The longitudinal axis of an object is the axis along which the length of an object is measured. There are a number of selectional restrictions imposed on the objects X and Y also. For example, the reason for the fourth selectional restriction can be gauged from the two phrases: across the road and along the road.

6.2.4 Processing a sentence

The sentence Put the block on the table can be used to show how Kalita's system obtains a meaning for a whole sentence from the meanings of its parts, i.e., the lexical entries of its constituent words.

The lexical entry for put specifies the achievement of a geometric relationship between an object and a location specified by a prepositional phrase. The meaning of the verb is specified in terms of a yet-unspecified geometric relation between two objects. The preposition on along with the objects involved leads to the sense that deals with support.

A bottom-up parser [FW83] returns the logical meaning representation as (put you block-1 (on block-1 table-1)). In this representation, the verb put takes three arguments: a subject, an object and the representation for a locative expression. Entities block-1 and table-1 are objects in the world determined to be the referents of the noun phrases. The logical representation has you as the value of the subject since the sentence is imperative.

Now, to obtain the intermediate meaning representation, the arguments of put in the logical representation are matched with the arguments in the following lexical entry for put:
put (l-agent, l-object, l-locative) —
  agent: l-agent
  object: l-object
  kernel-actions:
    geometric-constraint:
      execution-type: achieve
      geometric-relation: l-locative

This lexical entry has three arguments. After matching, l-agent has the value you, l-object has the value block-1, and l-locative has the value (on block-1 table-1). The value of the geometric-relation slot (of the kernel-actions slot in the representation) is filled in by the semantic representation for the l-locative argument which is created from the meaning of “on the table”, using the following definition of “on”:

on (X,Y) —
  geometric-relation:
    spatial-type: positional
    source-constraint-space: any-of (self-supporting-spaces-of (X))
    destination-constraint-space: any-of (supporter-surfaces-of (Y))

selectional-restrictions:
  horizontal-p (destination-constraint-space)
  equal (direction-of (normal-to
    destination-constraint-space), “global-up”)
  free-p (destination-constraint-space)

As a result, the intermediate meaning representation of “put the block on the table” is:

agent: you
object: block-1
kernel-actions:
  geometric-constraint:
    execution-type: achieve
    geometric-relation:
      spatial-type: positional
      source-constraint-space: any-of
        (self-supporting-spaces-of (block-1))
      destination-constraint-space: any-of
        (supporting-surfaces-of (table-1))
  selectional-restrictions:
    horizontal-p (destination-constraint-space)
    equal (direction-of (normal-to
      destination-constraint-space), “global-up”)
    free-p (destination-constraint-space)

In order to execute the action dictated by this sentence, the program looks at the knowledge stored about the block to find a part of the block on which
it can support itself. It observes that it can be supported on any one of its
takes and no face is more salient than any other. A cube (the shape of the
block) has six faces and one is chosen randomly as the support area. Next,
the program consults the knowledge stored about the table and searches for
a part or feature of the desk which can be used to support other objects. It
gathers that its function is to support "small" objects on its top. This top
surface is also horizontal. As a result, finally, the system concludes that one
of the sides of the cube has to be brought in contact with the top of the table.
The final meaning for the sentence obtained is

agent: you
object: block-1
kernel-actions:
  geometric-constraint:
    execution-type: achieve
    geometric-relation:
      spatial-type: positional
      source-constraint-space: block-1•side-2
      destination-constraint-space: table-1•top-1

block-1•side-2 represents a specific face of a specific block. table-1•top-1 repre-
sents the top surface of a specific table. This final representation is then
sent to a planner [JKBC91] which produces a plan for performing the task by
an animated agent in a given workspace. The plan is taken up by a simula-
tor [BWKE91] which establishes connection with Jack and then produces an
animation:

The block is initially sitting on top of a closed box. The agent
reaches for it with his right hand, grasps it, moves it to a point
near the top of a table to his left, places it on the table, and moves
his hand back.

As with Esakov's work, there were still unfortunate capability gaps in the
simulator available to Kalita. In particular, the lack of a flexible torso,
unchecked collisions with the environment, and no balance constraints led to
some painful-looking postures and object trajectories which passed through
obstacles.

6.2.5 Summary

This section has discussed the representation of meanings of some verbs and
prepositions, emphasizing the importance of geometric information such as
axes of objects, location of objects, distance or angle between objects, path of
object motion, physical contact between objects, etc., in the meaning represen-
tation of prepositions. Elsewhere it is shown that such geometric considera-
tions are important for not only representing verbs and prepositions, but
also adverbs [KB90].
In the work described here, the operational meanings of action verbs and their modifiers have been represented in terms of components pertaining to constraints and kinematic/dynamic characterization. For additional examples of decomposition see [Kal90].

6.3 Task-Level Simulation

The third experiment tested the feasibility of using what might be viewed as low-level task primitives to create task animations [Lev91]. If successful, this would have two advantages:

- Since we viewed some kind of low-level task primitives as being the output specification language of any language processing stages, it would allow us to design and test a set of primitives in parallel with the other system components.

- This kind of lower-level specification language might itself be usable by an engineer to generate animations in terms of task-level actions rather than having to specify particular body movements.

To illustrate the latter contrast, consider a scene with an animated agent, a table, and a cup on a shelf next to the table. The animator-engineer wants to create an animation of the agent moving the cup from the shelf to the table. A task-level specification could enable the animator-engineer to produce the desired behavior, using a set of task-action specifications. For example, the sequence

```
grasp-action (hand cup)
position-action (cup table-top)
```

could be used to generate an animation of the agent’s hand grasping the cup, followed by a positioning of the cup on the top of the table.

As a test environment we used an expanded version of some written instructions to remove a Fuel Control Valve (FCV) from an imaginary aircraft fuselage (Figure 6.1).

Fuel Control Valve Removal Instructions:

1. With right hand, remove socket wrench from tool belt, move to front of body. With left hand, reach to tool belt pocket, remove 5/8 inch socket, move to wrench, engage. Adjust ratchet for removal.

2. Move wrench to left hand bottom hole, apply pressure to turn in a loosening motion, repeat approximately 7 times to loosen threaded bolt.

3. Move wrench away from bolt, with left hand reach to bolt and remove bolt and washer from assembly, move left hand to belt pouch, place bolt and washer in pouch.

3Libby Levison.
Figure 6.1: A Frame from the Fuel Control Valve Removal Task. The FCV is the Cylindrical Object Mounted to the Flat Plate.

4. Move wrench to bottom right hand bolt, apply pressure to turn in a loosening motion, repeat approximately 7 times to loosen threaded bolt.

5. Repeat operation 3.

6. Move wrench to top bolt, apply pressure to turn in a loosening motion, repeat approximately 6 times to loosen threaded bolt. Move left hand to grasp assembly, loosen the bolt the final turn. Move wrench to tool belt, release. With right hand reach to bolt, remove bolt and washer, place in pouch. Return right hand to assembly, with both hands move Fuel Control Valve to movable cart and release.

6.3.1 Programming Environment

The work area, tools and parts for the scene were modeled with *Jack*. Just as the engineer who currently writes the instruction manuals has knowledge of the task and knows, for example, that a Phillips head screwdriver is required, it is assumed that the engineer-animator will have the knowledge required to lay out the scene of the animation. It is also assumed that a skilled engineer is already trained in analyzing tasks and developing instruction sets for the do-
main. This project simply provides a different medium in which the engineer can explain the task.

The task simulation is based on Yaps, a symbolic process simulator [EB90, BWKE91]. Yaps provides animation-directives which access Jack’s behaviors. These animation-directives are not only ordered and sequenced via Yaps’ temporal and conditional relationships [KKB88], but can also be composed to produce parameterized simulation procedures. These procedures, called task-actions, are defined for a number of parameters (agent, object, location, etc.). The same task-action can thus be used at various times with different parameters to create distinct animation segments. The possibility of defining and reusing these procedures simplifies the animation programming problem for the engineer. By extending these procedural compositions, high-level procedures could be generated so that the mapping from the instructions to these procedures would be straightforward.

KB [Esa90] is a frame-based, object-oriented knowledge system which establishes symbolic references to Jack’s geometric data. While Jack maintains and manipulates the geometric model of the world, KB maintains the symbolic information. Yaps uses KB’s symbolic representation to manipulate the geometric model. (These symbolic KB representations are passed to the Yaps task-actions as parameters.) This frees Yaps from “knowing” the specific world coordinates of an object or the object’s exact geometric representation. For instance, if Jack contains a model of a cup, KB would have an entry which identified cup as that particular Jack entity. Yaps has no knowledge of the object’s location; KB’s mapping from symbolic to geometric representation will resolve any ambiguity. Thus the animator need not talk about the-cup-on-the-table-at-world-coordinates-(x,y,z), but can reference the symbolic entity, cup. Should the cup move during the course of the action, KB resolves the problem of the cup’s exact location.

6.3.2 Task-actions

At the time of this research, Yaps provided only three low-level animation-directives with which to access Jack behaviors. These are generate-motion, create-constraint and delete-constraint. Generate-motion causes an object (not necessarily animate) to move from its current location to another. (No path planning was performed in the Jack version of the time, and Yaps handled frame-to-frame timing directly as described in Section 6.1.) Create-constraint establishes a physical link between two (not necessarily adjacent) objects. If two objects are linked together and one of the objects is moved, the second object moves along with it. The physical constraint (relation) between the objects is maintained. Create-constraint can be further specified to use positional and/or orientational alignments. Delete-constraint removes the specified constraint between two objects.

Yaps provides a mechanism for building animation templates by combining or composing the above animation-directives. Using different combinations of generate-motion, create-constraint, and delete-constraint, and vary-
ing the agents and the objects of these animation-directives as well as their temporal and causal relations, it is possible to build a set of task-actions. Task-actions can themselves be composed into more complex task-actions. As the procedures acquire more specification, the task-actions approach task-level descriptions. It is important to note, however, that task-actions simply define templates; an animation is realized by instantiating the task-actions, supplying parameters as well as timing constraints and other conditions. The composability of the task-actions allows for the definition of some abstract and high-level concepts. It is these high-level animation descriptions which will allow the engineer to program an animation at the task-level.

6.3.3 Motivating Some Task-Actions

The first templates to be defined were simply encapsulations of the Jack animation-directives: reach-action(agent object), hold-action(agent object) and free-object-action(object) – were just generate-motion, create-constraint and delete-constraint, respectively. (Although the names chosen for the task-actions do make some attempt to elicit their definition, there was no attempt to come up with definitive definitions of these actions in this segment of the research project.) In the following, the use of agent and object is simply for readability; for example, a hold-action can be applied between two objects (e.g., hold-action(wrench-head 5-8th-socket)).

Consider trying to describe the actions inherent in the example:

\textit{Move the cup to the table}

assuming that the agent is not currently holding the cup. The agent must first hold the cup before he can move it. How is this animation specified? Explicitly stating the sequence of actions:

\begin{verbatim}
reach-action agent cup
hold-action agent cup
\end{verbatim}

to cause the agent to reach his hand to the location of the cup and to constrain his hand to the cup seems awkward. Composing two task-actions allows a new task-action grasp-action to be defined:

\begin{verbatim}
(deftemplate grasp-action (agent object)
  reach-action (agent object)
  hold-action (agent object)).
\end{verbatim}

(This is the actual Yaps definition. Deftemplate is the Yaps command to define a new task-action template.) Grasp-action is a sequence of instantiations of two primitive task-actions.

Now that the agent can grasp the cup, how can he move the cup? A second action, position-action, is defined to relocate the cup to a new location and constrain it there:
(deftemplate position-action (object1 location)
   reach-action (object1 location)
   hold-action (object1 location)).

If a previous action had left an object (the cup) in the agent’s hand, this
task-action could be used to move the object to a new location (position-
action cup table). (In this instruction set, the only use of the instruction
“move something that is already being held” required that the object be
constrained to the new location. This is the justification of the hold-action
in this definition.) Note here that location could be the location of object2.

Thus, to animate the instruction:

\textit{Move the cup to the table}

the animation-script could be:

\text{grasp-action (agent-right-hand cup)}

\text{position-action (cup table-top)}.

It is still necessary to specify a list of commands, since no high-level task-
action has been defined for move, and therefore the action must be described
in increments. However move-action could be defined as:

\text{(deftemplate move-action (agent object1 location)}

\text{grasp-action (agent object1)}

\text{position-action (object1 location)}).

In other words, grasp (reach to and hold) object1, and position (move to and
constrain) object1 at location (where location might be the location of some
\text{object2}). In the \textit{Move the cup} example, the instantiation required to achieve
the desired animation would be:

\text{move-action (agent-right-hand cup table-top)}.

This conciseness is one benefit of task-action composition.

Once the cup is actually on the table, it can be “un-grasped” by using:

\text{free-object-action (cup)}

which breaks the constraint between the hand and the cup. If the hand is
later moved, the cup will no longer move with it.

The final animation script for \textit{Move the cup to the table} becomes:

\text{move-action (agent-right-hand cup table-top)}

\text{free-object-action (cup)}.

\section{Domain-specific task-actions}

The \textit{Move the cup to the table} example motivated a few fundamental task-
action definitions. Some of these are actions common to many instructional
tasks and milieus; this set of task-actions is also usable in the instruction set describing the FCV removal. However, it was also necessary to return to the instruction set and develop Yaps definitions for actions specific to the domain in question. These task-actions can be either primitive (see turn-action below) or compositional (see ratchet-action). The first new task-action, attach-action, is defined as:

```
(deftemplate attach-action (agent object1 object2)
  move-action (agent object1 object2)
  hold-action (object1 object2)).
```

This allows the agent to grasp object1, move it to the location of object2, and establish a constraint between object1 and object2. The expansion of this task-action is the command string:

reach-action, hold-action, reach-action, hold-action.

Attach-action could have been equivalently defined as:

```
(deftemplate attach-action (agent object1 object2)
  grasp-action (agent object1)
  position-action (object1 object2))
```

which would expand to exactly the same Jack animation-directive command string as above. The task-action definitions are associative; this provides flexibility and power to the system, and increases the feasibility of defining a minimal set of task-actions to be used throughout the domain.

The FCV removal instructions also require: turn-action (object degrees).

**Turn-action** causes the object to rotate by the specified number of degrees. The geometric definition of the object includes information on its DOFs; for example, around which axis a bolt will be allowed to rotate. At the time that this research was done, the system did not have a feedback tool to monitor Jack entities; instead of testing for an ending condition on an action (a bolt being free of its hole), actions had to be specified iteratively (the number of times to turn a bolt). Turn-action is actually a support routine, used in the final task-action needed to animate the FCV instructions: ratchet-action. This is defined as:

```
(deftemplate ratchet-action (object degrees iterations)
  turn-action (object degrees)
  turn-action (object –degrees)
  ratchet-action (object degrees iterations–1)).
```

Ratchet-action is used to animate of a socket wrench ratcheting back and forth.⁴

⁴Having to explicitly state a number of degrees is not an elegant programming solution; it would have been preferable to take advantage of Jack's collision detection algorithms to determine the range of the ratchet movement. Processing considerations at the time the work was done required this rather rough implementation.
The complete set of task-actions is listed below. With this set of only nine task-actions, it was possible to program the entire animation script from the natural language instructions (see Table 6.2 for an excerpt of the final animation script).

- reach-action (agent object)
- hold-action (agent object)
- free-object-action (object)
- grasp-action (agent object)
- move-action (agent object location)
- attach-action (agent object1 object2)
- position-action (object1 object2)
- turn-action (object degrees)
- ratchet-action (object degrees iterations)

6.3.5 Issues

Where Does Task-Action Decomposition Stop?

There is an interesting question as to whether, in defining task-actions, one needs to be concerned with variations that arise from differences in agents and their abilities.

Because our work is embedded in Jack, variations in agent ability at the animation specification level is not a concern. As long as the animation is within the agent’s capabilities (and thus the animation is “solvable”), substituting different agents gives different valuations of the tasks. By testing different agents with varying abilities, one can analyze the task requirements and gather information on human factors issues. Similarly, it is possible to vary workplace geometry, tools, and agent placement.

Note the comparison here between innate and planned action. In reaching to grab a cup, we do not think about how to control the muscles in the forearm; we do, however, consider the goal of getting our hand to the same location as the cup. This distinction between cognizant motion and action is internal in this animation; Jack manages the motor skills. The same distinction is found in the level of detail of the instructions. One does not tell someone:

\[
\text{Extend your hand to the cup by rotating your shoulder joint 40° while straightening your elbow joint 82° degrees. Constrain your hand to the cup by contracting fingers . . . .}
\]

Rather, we give them the goal to achieve and allow that goal to lend information as to how to accomplish the instruction. The hierarchy of the task-actions captures some of this knowledge.

The task-actions have been defined in such a way that they are not concerned with the abilities of a specific agent, but rather allow for interpretation
6.3. TASK-LEVEL SIMULATION

Table 6.2: Animation Script Excerpt.

;;; No. 1
;;; With right hand, remove socket wrench from tool belt,
;;; move to front of body. With left hand, reach to tool belt
;;; pocket, remove 5/8" socket, move to wrench, engage.
;;; Adjust ratchet for removal.

;;; with the right hand, grasp the wrench from the tool belt,
;;; and move it to site-front-body

(instantiate move-action
   (fred-rh wrench-handle fred-front-body-site planar)

   :instancename "r0-wrench-to-front"

   :time-constraints '((start now)
       (duration
         (eval (+ (fits fred-rh wrench-handle)
             (fits wrench-handle
                          fred-front-body-site))))))

;; with the left hand, attach socket to wrench handle.
;; an attach entails, reaching for the socket, grasping
;; it and moving it to the wrench head.
;; if successful, free the left hand from the socket.

(instantiate attach-action
   (fred-lh 5-8th-socket wrench-head
            attach-socket-time planar oriented)

   :instancename "r5-attach-socket"

   :time-constraints '((start (end "r0-wrench-to-front"))
       (duration (eval
         (+ (fits fred-lh 5-8th-socket)
             (fits fred-left-pocket
                          fred-front-body-site)
             attach-socket-time))))

:on-success !(progn
   (free-object-action fred-lh)
   (free-object-action 5-8th-socket)
   (hold-action wrench-head 5-8th-socket
                  :orientation-type '("orientation")))
based on each agent's capabilities. Not only does this allow the same animation script to be used for different agents, generating different analyses, but it also means that the definitions of the task-actions decomposition stops at the level of innate action. There is no need to have multiple task-action definitions for various physical attributes; Jack handles this issue for us.

**Instruction Translation**

We noted earlier that one advantage of a task-action level of specification was that it might allow an engineer/animator to animate tasks directly. In terms of the above task-actions, moving the cup to the table (noted in the introduction) could be animated by issuing either of two command sequences:

- grasp-action (agent-right-hand cup)
- position-action (cup table-top)
- free-object-action (cup)

or:

- move-action (agent-right-hand cup table-top)
- free-object-action (cup).

In both cases, the engineer has decided to release the constraint between the agent and the cup as soon as the cup is on the table-top. The engineer has also described the required animation at the task-level.

**Sequencing Sub-tasks**

**Yaps** is a simultaneous language; that is, all task-action instantiations are resolved concurrently. To sequence the actions and force them to occur in a specific order, the engineer/animator must use the **timing-constraints** option provided by **Yaps**. This construct allows the user to specify starting, ending and duration conditions for the instantiation of each action. It is possible to achieve the ordering needed to create a sequential animation by predicating the starting condition of instruction-2 on the ending condition of instruction-1; but a task-action template, which is defined as a series of other task-actions, has the sequencing automatically built in via the instantiation process. If this were not the case, defining grasp-action, for example, would be impossible because achieving and completing the reach-action before starting the hold-action could not be guaranteed.

The actions do not need to be performed discretely. Other **Yaps** timing constructs allow the actions to be overlapped and delayed by specifying (start (after 5 min)) or (start now), for example [KKB88]. Nor is defining a discrete linear order on the sub-tasks the only possibility. The simultaneous nature of **Yaps** is used to animate actions (such as moving an object with both hands) by simultaneously animating:

- move-action (agent-left-hand box)
- move-action (agent-right-hand box).
6.4. A Model for Instruction Understanding

The Yaps timing constraints provide a powerful mechanism for specifying the relationships among the task-actions in the animation. Timing is one of the most critical issues involved in generating realistic animations; the power that Yaps provides in resolving timing issues greatly enhances the potential of the Jack animation system.

Task Duration

The Yaps timing constraints provide a powerful mechanism for specifying the inter-relationships among the task-actions in the animation script. Timing is one of the most critical issues involved in generating realistic animations. We have already noted that it is not sufficient to simply list all the actions; they must be times, sequenced and connected temporally. As in Esakov’s work, adaptations of Fitts’ Law were used to determine minimum action times. Fitts’ Law was used to calculate the duration of all reach-action instantiations. Thus, time requirements were cumulative (i.e., the sum of the sub-task-action times). Create-constraint uses a small default constant time to estimate sub-task duration. Although Fitts’ Law only approximates the action times in this domain and must be further scaled by a motivation factor, it does give reasonable estimates. Relative to one another, the sub-task times make sense. Although the length of each task-action might not be correct, the animation does appear to be temporally coherent.

6.3.6 Summary

Recent work in defining animation behaviors reviewed earlier in this book greatly expands the set of animation directives available in Jack. In our current work, we will investigate using the new animation behaviors to script animations. Since animation directives form the semantical basis for our action definitions, a more powerful set of animation directives provides us with a richer language with which to work. As it becomes easier to define new task-actions, the animator will spend less time coordinating sub-actions.

Finally, this new vocabulary will allow us to express tasks (or define task-actions) which differ from the earlier work in their semantic content. Our first attempt at rescripting the instruction set resulted in a more realistic animation, in that the new behaviors allowed us to include such low-level actions as take step to maintain balance when the animated agent was reaching beyond his comfort range. We need to compare the expressive powers of the previous animation directives with the enhanced set of animation behaviors.

6.4 A Model for Instruction Understanding

The three experiments described in the previous sections were all concerned with the operational semantics of single-clause commands. But the range of

Barbara Di Eugenio, Michael White, Breck Baldwin, Chris Ceb, Libby Levison, Michael Moore.
tasks that can be communicated to an agent with such commands is very limited — the less expertise and experience on an agent’s part, the more he needs to be told. A telling example of this is given in [Pri81]. Here, Prince compares a recipe for stuffed roast pig given in a nineteenth century French cookbook with that given in Rombauer’s contemporary *The Joy of Cooking*. The former says, essentially, “Roast pig. Stuff with farce anglaise.” Rombauer’s instructions go on for two pages: she assumes very little culinary experience with pigs on the part of today’s men and women.

Multi-clause commands are very common in maintenance and assembly instructions, such as the following examples from Air Force maintenance manual T.O. 1F-10C-2-94JG-50-2:

> “With door opened, adjust switch until roller contacts cam and continuity is indicated at pins A and B. Verify positive switch contact by tightening bottom nut one additional turn.” (p. 5-24)

> “Hold drum timing pin depressed and position entrance unit on drum. Install three washers and three bolts, release drum timing pin, and torque bolts to 60-80 inch-pounds.” (p. 6-14)

Now just as multi-clause texts are commonly organized into paragraphs, multi-clause instructions are commonly organized into steps. In fact, the above multi-clause commands are actually single steps from longer, multi-step instructions. While there are no firm guidelines as to what a single instruction step should encompass, there is a strong tendency at least for steps to be organized around small coherent sub-tasks (such as adjusting a switch or installing a component, as in the above examples). A typical step may specify several actions that need to be performed together to accomplish a single sub-task, or several aspects of a single complex action (e.g. its purpose, manner, things to watch out for, appropriate termination conditions, etc.). The agent must develop some degree of understanding of the whole step before starting to act.

In our current work on instruction understanding, we add to this sub-task sense of step, the sense that a step specifies behavior that the agent must attend to continuously: while carrying out a step, the agent’s attention is fixed on the task at hand. Communication with the instructor is not allowed until completion (or failure) of the current step. Because of this, a step defines the extent of the instructions that must be processed before the agent begins to act on them. (With some reflection on one’s own confrontations with new instructions, it is easy to recall situations where one has tried to understand too much or to act on too little understanding. It is not always obvious when one should begin to act.)

While our focus is on multi-clause instructions, it turns out that many of their important features can be demonstrated simply with two-clause instructions. (As in many things, the biggest leap is from one to two.) The two-clause example we will use here to describe our framework for instruction understanding and animation is:
6.4. A MODEL FOR INSTRUCTION UNDERSTANDING

"Go into the kitchen to get me the coffee urn."

This example will be used to illustrate, among other things:

- expectations raised by instructions;
- the need for incremental generation of sub-goals (plan expansion) in order to act in accordance with instructions;
- the need to accommodate the agent's behavior in carrying out actions, to the objects being acted upon; and
- the need to develop plans at more than one level.

Figure 6.2 shows a schematic diagram of the AnimNL (ANIMation from Natural Language) architecture. Before going through the example, we want to call attention to the system's overall structure -- in particular, to the fact that it consists of two relatively independent sets of processes: one set of which produces commitments to act for a particular purpose, what we call animated task actions -- e.g.

- \texttt{goto(door1, open(door1))} -- "go to door1 for the purpose of opening it"
- \texttt{grasp(urn1, carry(urn1))} -- "grasp urn1 for the purpose of carrying it"

and the other set of which figures out how the agent should move in order to act for that purpose. In this framework, instructions lead to initial commitments to act, and actions once embarked upon allow further commitments to be made and acted upon. (While our discussion here will be in terms of single-agent procedures, it can be extended to multi-agent procedures by adding communicative and coordinating actions. As shown in earlier chapters, both Jack and its behavioral simulator can support the activity of multiple agents. However, extending the upper set of processes to delineate the communication and coordination required of multiple agents cooperating on a task requires solution of many problems currently under investigation by members of the AI planning community.

We now begin by giving AnimNL the instruction step:

"Go into the kitchen to get me the coffee urn."

A picture of the agent in its starting situation, when it is given the instruction, is shown in Plate 6.

Steps are first processed by a parser that uses a combinatory categorial grammar (CCG) [Ste90] to produce an action representation based on Jackendoff's Conceptual Structures [Jac90]. We are using CCG because of its facility with conjoined constituents, which are common in instructions -- for example

"Clear and rope off an area around the aircraft and post warning signs." [Air Force Maintenance manual T.O. 1F-16C-2-94JG-50-2]
Figure 6.2: AnimNL System Architecture.
6.4. A MODEL FOR INSTRUCTION UNDERSTANDING

We are using Jackendoff's Conceptual Structures for two reasons: first, the primitives of his decompositional theory capture important generalizations about action descriptions and their relationships to one another, and second, they reveal where information may be missing from an utterance and have to be provided by inference. For the instruction step "Go into the kitchen to get me the coffee urn", the parser produces the following structure:

\[
\begin{align*}
& \text{GO}_{SP}[\text{[AGENT]}, \text{[TO}([\text{IN}([\text{KITCHEN}])])]])_\alpha \\
& \text{FOR}(\beta) \\
& \text{CAUSE}(\gamma, \text{GO}_{SP}([\text{COFFEE-URN}][j], [k]))_\beta \\
& \text{FROM}([\text{AT}(j)]))_\gamma \\
& \text{TO}(l) 
\end{align*}
\]

This representation makes explicit the fact that getting the coffee urn involves its moving from its current location to a new one (which should be the location of the instructor). The FOR-function (derived from the to-phrase) encodes the purpose relation holding between the go-action \(\alpha\) and the get-action \(\beta\). Indices indicate different instances of a single conceptual type [ZV92].

From these indexed conceptual structures, an initial plan graph is constructed to represent the agent's intentions, beliefs and expectations about the task it is to perform. To do this, the system consults the agent's knowledge of actions and plans (the Action KB and Plan Library in Figure 6.2), to develop hypotheses about the instructor-intended relationships between the specified actions (e.g., temporal relations, enablement relations, generation relations, etc.). The initial plan graph for our running example is shown in Figure 6.3.

This initial plan graph is further elaborated through processes of reference resolution, plan inference, reference grounding, plan expansion and performance (through simulation). To show the interaction between these processes and how they are used to elaborate the plan graph, we will contrast our example

"Go into the kitchen to get me the coffee urn."

with a somewhat different but related example

"Go into the kitchen and wash out the coffee urn."

In the first case, recall from the conceptual structure produced by the parser, that "get me" is interpreted as an instance of a "cause something to go somewhere" action. One recipe that the system has in its Plan Library for accomplishing this is shown in Figure 6.4. With respect to this recipe, "go" can be seen as a substep of "get" — which is one way it can serve the purpose of "get". (This action representation and the plan graph are described in greater detail in [EW92].)

Getting an object from one place to another requires first going to its location. This leads to the assumption, noted in Figure 6.3, that the coffee
urn is in the kitchen. (The role of plan inference in instruction understanding is discussed in more detail in [Di 92, DW92].) Reference resolution cannot contribute any further constraints to the description “the coffee urn”, since (1) there is no urn in the discourse context (nor anything that has a unique coffee urn associated with it), and (2) the assumption that the urn is in the kitchen is incompatible with its being unique in the current spatio-temporal context (which is the room next to the kitchen). Reference grounding does not attempt to associate this description with an object in the current spatio-temporal context, for the same reason. In fact, the agent will not attempt to ground this referring expression until it has entered the kitchen. (Whether the agent then succeeds immediately in grounding the expression will depend on whether the urn is perceivable – i.e., out in full view. We will discuss this shortly. In any case, the agent expects to be able to get access to the urn when it gets to the kitchen. This is what will drive it to seek the urn, if it is not in view when it gets to the kitchen.)

In the contrasting example “Go into the kitchen and wash out the coffee urn”, the system again hypothesizes that the purpose relation between go and wash-out is a substep relation – but in this case, it is because washing out an object requires being at a washing site (e.g., a sink or tub). That kitchens usually have sinks gives further weight to this hypothesis.

Reference resolution may now contribute something to the agent’s understanding of the definite expression “the coffee urn”. While the discourse-context does not provide evidence of a unique coffee urn, either directly or by association, there is also no evidence against the hypothesis that the urn is in the current spatio-temporal context. An initial hypothesis added by reference resolution that the urn is in the current space, if confirmed by reference
6.4. A MODEL FOR INSTRUCTION UNDERSTANDING

![Table]

<table>
<thead>
<tr>
<th>Header</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CAUSE([AGENT], [GOSp(i, k)])]</td>
</tr>
<tr>
<td>FROM([AT(j)])</td>
</tr>
<tr>
<td>TO(l)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>[GOSp(i, TO([AT(j)])])_1</td>
</tr>
<tr>
<td>[CAUSE(i, [GOCh(j, [TO([AT(l)])])])_2</td>
</tr>
<tr>
<td>[GOSp(i, k)]_3</td>
</tr>
<tr>
<td>[WITH(j)]_3</td>
</tr>
</tbody>
</table>

- Annotations -
  - \( \gamma_1 \) enables \( \gamma_2 \) enables \( \gamma_3 \)

<table>
<thead>
<tr>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[NOT BESp(j, l)]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>[BESp(j, l)]</td>
</tr>
</tbody>
</table>

Figure 6.4: A Move Something Somewhere Action.

The next thing to discuss is how the plan graph is expanded, and why it is expanded incrementally, as actions are performed in accordance with earlier elements of the plan graph. How it is expanded is through subgoal generation down to what we have called annotated task actions. This process makes use of a new kind of planner that (1) eschews pre-conditions in favor of decisions based on the agent’s positive and negative intentions, and (2) takes upcoming intentions into account when deciding how to expand current goals, so as to put the agent in the best position with respect to satisfying those intentions. This planner, called ItPlan3, is described in more detail in [Gei92]. It is also
the source of the annotations of purpose in annotated task actions.

The main reason why the plan graph is expanded incrementally is that the agent does not have sufficient knowledge, before beginning to act, of what it will need to do later. In particular, AnimNL assumes that an agent cannot have up-to-date knowledge of any part of its environment that is outside its direct perception. (An AnimNL agent may know what non-visible parts of its environment were like, when it saw them earlier, and have expectations about what they will be like, when it sees them next, but its knowledge is limited to general truths about the world and to its direct perceptions.) As for the extent of the agent's perception, it is assumed that an agent cannot see into any space that has no portal open into the space the agent occupies. Thus AnimNL agents have to open doors, closets, boxes, etc., if they want to know what is inside, or go into other rooms to find out what is there.

What this means in our example is that only the plan graph node corresponding to "go into the kitchen" can be expanded – in this case, to "go over to the door", "open door", and "enter kitchen" – before the agent begins to act. The node corresponding to "go to the location of the coffee urn" cannot be expanded until the door has been opened and the agent can see whether or not the urn is visible. If it is visible, the agent can go to its location (Plate 6). If it is not visible, this same node must be expanded with actions corresponding to finding the urn – going through the kitchen cabinets one at a time looking for the urn, until it is found or all cabinets have been searched (Figure 6.5).

When an annotated task action becomes sufficiently specified for the agent
to be ready to commit to it and temporal dependencies permit such commitment, it is gated, triggering other, low-level planning processes (see Figure 6.2 below the “action gate”). An annotated task action is sufficiently specified if

- the action is “executable” (i.e., a task action, as described in Section 6.3).

- all actions temporally prior to it have been committed to. (Note that previous actions need not be completed before a new action is committed to: an agent can be (and usually is) doing more than one thing at a time.)

- its purpose has been determined.

It is worthwhile saying a bit more here about these purpose annotations, since we have come to believe they play a large part in low-level decisions about how to act. The kind of observations that motivates them are the following:

- when told to pick up a book and hand it to someone, an agent will grasp it one way;

- when told to pick up the same book and turn it over, an agent will commonly grasp it in quite a different way;

- when told to pick up the book and open to page 70, the agent will grasp it yet a third way.

- when just told to pick up the book, and nothing further, agents commonly grasp it, lift it up and wait expectantly for the next command.

These variations in grasp extend to such low-level features as grasp site and wrist position.

What we have tentatively concluded from such observations is that when agents don’t know the purpose of some action they are told to perform, they put themselves into a position that easily supports subsequent action. Of course, always going into a position in which an agent is poised for subsequent action is very inefficient, especially when the agent knows what that subsequent action will be. In that case, he or she acts in such a way to smoothly and efficiently transition from one to the other. In AnimNL, purpose annotations (including “PFA” or poised for action) are there to allow the simulator, upon action commitment, to come up with the most effective ways of moving the agent’s body for the given purpose. It is also why the system is designed to delay commitment until it knows the purpose of any task action or knows that the only thing it can know is PFA.

When an action is committed to, there is still further work to be done in order to determine the agent’s behavior. In particular, one result of the experiment described in the previous section (Section 6.3) was our recognition of the need for tailoring an agent’s behavior in carrying out an action to the
type of object given as an argument to that action. This follows from the fact
that the same Natural Language verb is commonly used with different objects
to denote very different behavior on an agent's part, and for a task animation
to be correct, these differences must be depicted.

Consider, for example, the following definition (from [JCM73]) of the
word "remove" and sentences illustrating its use:

**Remove:** to perform operations necessary to take an equipment unit
out of the next larger assembly or system;
to take off or eliminate; to take or move away.
1a. Remove bleed air shutoff valves.
1b. Remove bolts from nuts.
2. Remove paint.
3. Remove covers.

For each different object, the behavior needed to effect a "remove" is quite
different. The question is whether to define a single remove-action, to use
in animating both Remove the paint and Remove the bolt? The alternative —
defining a multitude of animation procedures (e.g. remove-paint, remove-bolt,
remove-nut, remove-nail, remove-boxtop, etc.) — appears expensive in terms
of time and effort, and prone to error.

The solution we are adopting is to build a hybrid system. Instead of
specifying complete definitions for each verb, we can identify the core or kernel
action for a verb like remove in a fashion similar to that described in
Section 6.2. We will use this core meaning, central to many different instantiations of the verb, in building the task-action. The missing information can
be supplied by the verb's object: The knowledge base is object-oriented and
so can store relevant information about individual objects. For example, one
slot of information might be the DOFs an object has — a bolt "knows" (i.e., its
gemetric constraints specify) around which axis it turns. Joint and rotation
information is already available in Jack.

The hybrid system would process an instruction by combining the informa-
tion in the two representations — the underspecified definitions of the
task-actions, in conjunction with the object-oriented knowledge base. By
identifying which information is lacking in the task-actions, the system can
try to supply that information from the knowledge base.

The advantages of a hybrid system is economy of both action definitions
and of the feature information to be stored. We no longer need to worry
about developing separate definitions for each animation movement based on
distinct verb/object pairs. Instead we take advantage of the compositional
nature of the task-actions, and the object-oriented, hierarchical knowledge-
base. Using these utilities, we can define a single animation definition for
remove which will allow us to animate both Remove the bolt and Remove nuts
from bolts while still distinguishing the instruction Remove covers.

Our work on using complex Natural Language instructions to motivate the
behavior of animated agents is still in its infancy. There is much more to be
done before it is a useful tool in the hands of task designers and human factors
engineers. On the other hand, we have begun to demonstrate its potential flexibility in accommodating the task behavior of an agent to the environment in which the task is being carried out and the agent's own capabilities.