1.
Lifelike Computer Characters: the Persona project at Microsoft Research

Gene Ball, Dan Ling, David Kurlander, John Miller, David Pugh, Tim Skelly, Andy Stankosky, David Thiel, Maarten Van Dantzich and Trace Wax

Microsoft Research

ABSTRACT

This article describes the Persona project at Microsoft Research, which is exploring *social user interfaces* that employ an explicitly anthropomorphic character to interact with the user in a natural spoken dialogue. The prototype system described here integrates spoken language input, a simple conversational dialogue manager, reactive 3D animation, speech output and sound effects to create *Peedy the Parrot,* a conversational assistant who accepts user requests for audio CDs and then plays them.

1.1 INTRODUCTION

The computing industry of the 90's is in the process of fully adopting the graphical user interface metaphor pioneered by Xerox PARC in the 70's. This metaphor, first explored by the Smalltalk system on the Alto [8], was already firmly defined in most significant respects when the Xerox Star was introduced in 1980 [21]. The concepts of WYSIWYG editing, overlapping screen windows, and the direct manipulation of system objects as icons had all been thoroughly demonstrated. The subsequent decade has seen considerable refinement of the original ideas, particularly regarding usability issues and the idea of visual affordances [18], but the essence of the original metaphor is intact. As GUIs become the industry standard, it is appropriate to look ahead to the next major metaphor shift in computing. While there are undoubtedly many further improvements that can (and will) be made to the GUI metaphor, it seems unlikely that computing in 2015 will still be primarily a process of clicking and dragging buttons and icons on a metaphorical desktop [17]. Improvements in display technology, miniaturization, wireless communication, and of course processor performance and memory capacity will all contribute to the rapid proliferation of increasingly sophisticated personal computing devices. But it is the evolution of software capability that will trigger a basic change in the user interface metaphor: computers will become assistants rather than just tools.

The coming decade will see increasing efforts to develop software which can perform large tasks autonomously, hiding as many of the details from the user as possible. Rather than invoking a sequence of commands which cause a program to carry out small, well-defined, and predictable operations, the user will specify the overall goals of a task and delegate to the computer responsibility for working out the details. In the specification process the user will need to describe tasks rather than just select them from predefined alternatives. Like a human assistant, the machine may need to clarify uncertainties in its understanding of the task, and may occasionally need to ask the user's advice on how best to proceed. And like a human, it will make suggestions and initiate actions that seem appropriate, given its model of the user's goals. Finally, a successful assistant will sometimes take risks, when it judges that the costs of interrupting the user outweigh the potential costs of proceeding in error.

1.1.1 Requirements for an Assistive Interface

The machine-like metaphor of a direct manipulation interface is not a good match to the communication needs of a computer assistant and its boss. In order to be successful, an assistant-like interface will need to:

- Support interactive give and take. Assistants don’t respond only when asked a direct question. They ask questions to clarify their understanding of an assignment, describe their plans and anticipated problems, negotiate task descriptions to fit the skills and resources available, report on progress, and submit results as they become available.

- Recognize the costs of interaction and delay. It is inappropriate to require the user’s confirmation of every decision made while carrying out a task. Current systems usually ask because they have a very weak understanding of the consequences of their actions. An assistive interface must model the significance of its decisions and the potential costs of an error so that it can choose to avoid bothering the user with details that aren’t important. Especially as the assistant becomes responsible for ongoing tasks, the cost of interrupting a user who is concentrating on something else (or of waiting when the user isn’t available), must be taken into account.

- Manage interruptions effectively. When it is necessary to initiate an interaction with the user, the assistant needs to do so carefully, recognizing the likelihood that the user is already occupied to some degree. The system may be able to tell that the user is typing furiously, or talking on the telephone, and should wait until an appropriate pause (depending on the urgency of the interruption). Even when apparently idle, the user might be deep in thought, so a non-critical interruption should be tentative in any case.

- Acknowledge the social and emotional aspects of interaction. A human assistant quickly learns that “appropriate behavior” depends on the task, the time of day, and the boss’s mood. To become a comfortable working partner, a computer assistant will need to vary its behavior depending on such variables as well. Social user interfaces have tremendous potential to enliven the interface and make the computing experience more enjoyable for the user, but they must be able to quickly recognize cues that non-critical interactions are not welcome.

1.1.2 Conversational Interfaces: the Persona project

How will we interact with computer assistants? The most natural and convenient way will be by means of a natural spoken dialogue. Since we are convinced that users will be unwilling to speak to the computer in specialized command languages, spoken conversational interaction will only become popular when the assistant can understand a broad range of English paraphrases of the user’s intent. However, sufficient progress has now been made on speech recognition and natural language understanding that the prospect of a useful conversational interface has become a realistic goal.

The Persona project at Microsoft Research began in late 1992 to undertake the construction of a *lifelike* computer assistant, a character within the PC which interacts with the user in a natural spoken dialogue, and has an expressive visual presence. The project set out to build on the ongoing research efforts at Microsoft in speech recognition [11] and natural language processing (NLP) [12], as well as developing new reactive three-dimensional computer animation techniques [2]. The goal was to achieve a level of conversational competence and visual reactivity that allows a user to suspend disbelief and interact with our assistant in a natural fashion.

As a first step, we have constructed a prototype conversational system in which our character (a parrot named Peedy) acts as music assistant, allowing the user to ask about a collection of audio CD's and select songs to be played. Peedy listens to spoken English requests and maintains a rudimentary model of the dialogue state, allowing him to respond (verbally or with actions) in a conversationally appropriate way.

1.1.3 Related Work

The creation of a lifelike computer character requires the integration of a wide variety of technologies and skills. A comprehensive review of all the research relevant to the task is therefore beyond the scope of this chapter. Instead, this section simply attempts to provide references to the work which has most directly influenced our efforts.

The work of Cliff Nass and Byron Reeves at Stanford University [16] has demonstrated that interaction with computers inevitably evokes human social responses. Their studies have shown that in many ways people treat computers *as human*, even when the computer interface is not explicitly anthropomorphic. Their work has convinced us that since users will anthropomorphize a computer system in any case, the presence of a lifelike character is perhaps the best way to achieve some measure of control over the social and psychological aspects of the interaction.

The Microsoft “Bob” [15] product development team has created a collection of home computer applications based entirely on the metaphor of a *Social User Interface*, in which an animated personal guide is the primary interface to the computer. The guide communicates to the user through speech balloons which present a small group of buttons for the operations most likely to be used next. This allows the user to focus on a single source of relevant information without becoming overwhelmed by large numbers of options. The guides also provide tips and suggestions to introduce new capabilities, or to point out more efficient ways of completing a task. User studies with Bob have verified that for many people, the social metaphor reduces the anxiety associated with computer use.

Efforts to create lifelike characters are underway in a number of other research organizations, including the Oz project at Carnegie-Mellon University [3], Takeuchi's work at Sony Computer Science Laboratory [22], the Jack project at the University of Pennsylvania [1], the CAIT project at Stanford [9], and the Autonomous Agents Group at the M.I.T. Media Laboratory [14].

In the linguistic processing required of a conversational assistant, we attempt to find a practical balance between knowledge intensive approaches to understanding (e.g. Lockheed's Homer [23]) and more pragmatic natural command languages (e.g. CMU's Phoenix [25]).

We are convinced that useful conversational interfaces will have to simulate many of the subtle dialogue mechanisms that humans use to communicate effectively. Our (still very preliminary) efforts in that direction are based on the work of Cohen [6], Clark, [5] and Walker [24].

Relevant references on the visual presentation of a character include work on physically realistic animation at Georgia Tech [10] and DEC [26], procedural generation of natural motion at NYU [19], and the coordination of simulation and animation at IBM [13]. Our work on pre-compiled action plans is most similar to the work of Schoppers [20]. Key issues for the effective audio presentation of lifelike characters include work on emotive speech [4] and rich soundscapes [7].

1.2 PERSONA SYSTEM OVERVIEW

For the reasons discussed above, it seems quite likely that conversational assistants will play a major role in our interactions with computers in the next century. Many of the technologies involved, including speech recognition, natural language understanding, animation, and speech synthesis, have been the focus of significant research efforts for many years. In addition to specialized research efforts in those topics, we decided in 1992 to undertake the construction of a complete conversational assistant. That decision was motivated by two complementary goals:

- First, an integrated system could serve as a testing ground for the individual technologies. The requirements of a conversational assistant would stress each technology in specific (and sometimes unexpected) ways, and serve to motivate and guide research for those components. Further, many integration issues will have to be resolved before conversational assistants can become a mainstream capability, and a prototype system can be a productive way to explore methods for combining complex technologies into a coherent architecture.
- Secondly, the overall experience of interacting with a computer assistant is likely to be profoundly different from using the component technologies individually. The anthropomorphic nature of the assistant ensures that it will generate social and psychological responses in the user which are qualitatively different from those encountered with traditional computer interfaces. In addition, the use of spoken conversation is likely to raise expectations of human competence that must be controlled (i.e. lowered) in order to avoid disappointing the user. We expected that a conversational prototype would be a useful testbed for exploring the dynamics of interaction with a computer character-- dynamics that can't be experienced without an integrated system.

A diagram of the prototype system that we built (named Personal Digital Parrot One: i.e., PDP1, or Peedy for short) can be seen in Figure 1. Because of the anthropomorphic nature of the system, the name Peedy naturally transferred to our initial character (a parrot) as well. In the remainder of the chapter, "Peedy" (or "he") will be used to refer to the prototype system and the character interchangeably.

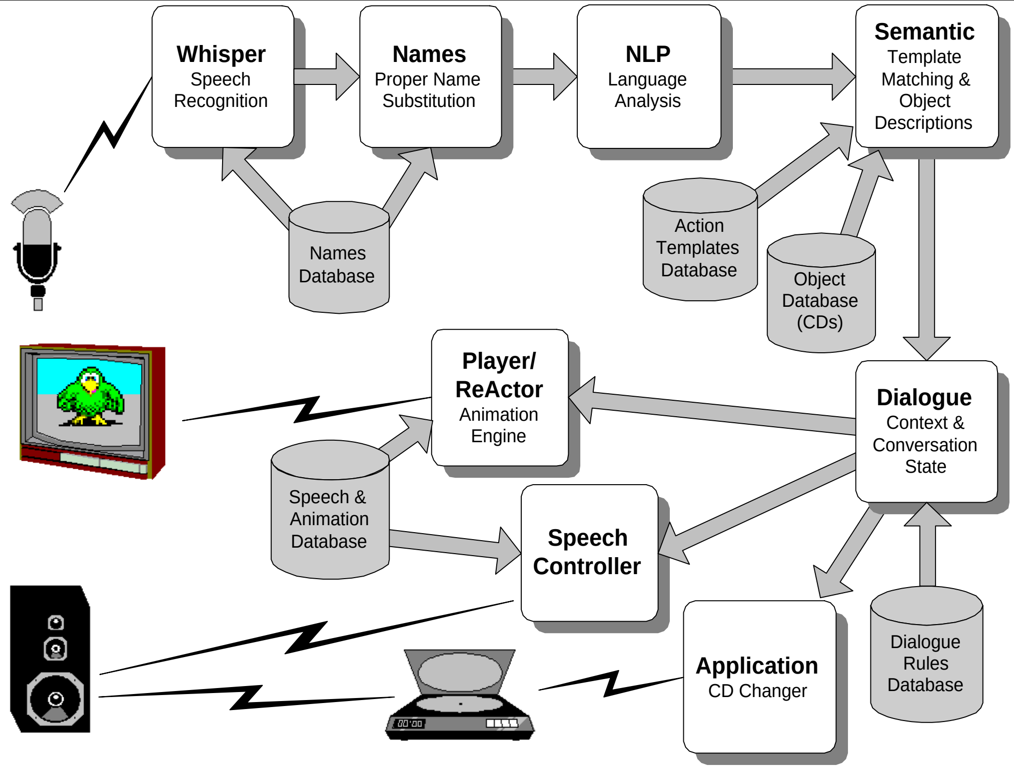

Figure 1: System diagram of the Persona conversational assistant

Each time Peedy receives a spoken input, he responds with a combination of visual and audio output. Figure 2 shows a transcript of a brief interaction with Peedy. For purposes of discussion here, the system will be split into three sub-systems:

- *Spoken language processing*, (consisting of the Whisper, Names, NLP, and Semantic modules in Figure 1) which accepts microphone input and translates it into a high level input event description,
- *Dialogue management*, (Dialogue in Figure 1) which accepts input events and decides how the character will respond,
- *Video and Audio output*, (Player/ReActor & Speech Controller) which, in response to dialogue output requests, generates the animated motion, speech and sound effects necessary to communicate to the user in a convincingly lifelike way.

These sub-systems constitute the user interface of the system, which controls a simple application that allows the user to select and play music from a collection of audio compact discs (labeled Application in Figure 1).

1.2.1 Goals for the subsystems

In each of these three areas, we began the prototype with a number of long term goals in mind, and then tried to achieve a minimum workable subset on a realistic path toward those goals. In this section, we enumerate those goals and summarize the prototype's status with respect to them. Discussion of future work has been deferred to the end of the chapter.

Language: Our eventual goal for the spoken language subsystem is to allow users to express requests in natural conversational English, without any need to learn a specialized command language. The character should be able to understand any likely paraphrase of a request that is within its capabilities.

In the current prototype, we have tried to construct a framework that could be extended to meet that goal, but its current capabilities are quite limited. Spoken commands must currently come from a limited set of about 150 "typical" utterances that might be encountered in the CD audio application. These utterances are recognized as paraphrases of one of 17 canonical requests that Peedy understands.

Dialogue: The dialogue controller is probably the most open-ended component of the system. Since it acts as Peedy's "brain", deciding how to respond to perceptual stimuli, it could eventually become a quite sophisticated model of a computer assistant's memory, goals, plans, and emotions. However, in order to reduce complexity, we decided to limit ourselves to "canned plans" -- e.g. predefined sequences of actions that can be authored as part of the creation of a character, then activated in response to input events. This mechanism must be made flexible enough to allow multiple sequences to be active simultaneously (e.g. to let a misunderstanding correction sub-dialogue occur at any point within a music selection interaction). In addition, to enhance the believability of a character, we feel that its behavior should be affected by memories of earlier interactions within the dialogue (or in previous conversations) and by a simple model of its emotional state.

The dialogue controller in the current system includes sequences for only a few conversational interactions, with no facility for managing sub-dialogues. We have experimented with some preliminary implementations of episodic memory and an emotional

	[Peedy is asleep on his perch.]
User:	Good morning, Peedy.
	[Peedy rouses]
Peedy:	**Good morning.**
User:	Let's do a demo.
	[Peedy stands up, smiles]
Peedy:	**Your wish is my command, what would you like to hear?**
User:	What have you got by Bonnie Raitt?
	[Peedy waves in a stream of notes, and grabs one as they rush by.]
Peedy:	**I have "The Bonnie Raitt Collection" from 1990.**
User:	Pick something from that.
Peedy:	**How about "Angel from Montgomery"?**
User:	Sounds good.
	[Peedy drops note on pile]
Peedy:	**OK.**
User:	Play some rock after that.
	[Peedy scans the notes again, selects one]
Peedy:	**How about "Fools in Love"?**
User:	Who wrote that?
	[Peedy cups one wing to his 'ear']
Peedy:	**Huh?**
User:	Who wrote that?
	[Peedy looks up, scrunches his brow]
Peedy:	**Joe Jackson**
User:	Fine.
	[Drops note on pile]
Peedy:	**OK.**

Figure 2: Sample dialogue with Peedy

model, but haven't fully integrated those with the rest of the system.

Video and Audio output For the animation subsystem, our goal is to create a convincing visual and aural representation of a character, which when given fairly abstract requests for action by the dialogue controller, can then carry out those requests with smoothly believable motion and synchronized sound. Because the character's actions must fit into the ongoing dialogue, the ability to instantly produce an appropriate animation is critical. We also wish to use film techniques to enhance the clarity and interest of the visual presentation and to create a rich and convincing acoustic environment. Finally, some variation is needed in the animation sequences so as to avoid obvious repetition and maintain the illusion of natural motion.

The Player/ReActor runtime animation system has been very successful at producing reactive real-time sequences of high quality animation. In the current system, however, all camera control and movement variability must be hand authored. We also chose to forego the flexibility of a general text-to-speech system because such systems currently lack the naturalness and expressivity that our character requires. Thus in the current system, the authoring effort required to produce new animation sequences (defining character motion, camera control, sound effects, and pre-recorded speech) is much higher than we would like.

1.2.2 Hardware Environment

The language and dialogue subsystems of the Peedy prototype currently run on a 90 MHz Pentium PC under Windows NT, without any specialized signal processing hardware. ReActor (including graphics rendering at 8 to 15 frames per second) runs on a Silicon Graphics Indigo2. The system is coded in G (language transformation rules), C, C++, and Visual Basic. Language processing for each utterance, exclusive of database searches, typically takes well under a second. However, communication delays between system components, and database queries increase the typical response latency to several seconds. While much of this delay can be attributed to the prototyping development environment, we expect the reduction of system latency to be a major ongoing challenge.

1.3 IMPLEMENTATION

This section describes the capabilities and implementation of each sub-system of the Persona prototype in detail. The prototype is quite shallow in its capabilities, yet it is very effective at producing the illusion of conversational interaction. The implementation specifics therefore serve both to document the shortcuts and tricks that we've used to achieve that illusion, and also to demonstrate that the system organization can support continued development toward the goals outlined above.

The final section outlines the next steps that we feel are appropriate for each component, and discusses our plans for continued development.

1.3.1 SPOKEN LANGUAGE PROCESSING

As described above, a key goal for the spoken language subsystem of the Persona project is to allow users flexibility to express their requests in the syntactic form they find most natural. Therefore, we have chosen to base the interface on a broad-coverage natural language processing system, even though the assistant currently understands requests in only a very limited domain.

It is precisely the flexibility (and familiarity) of spoken language that makes it such an attractive interface: users decide what they wish to say to the assistant, and express it in whatever fashion they find most natural. As long as the *meaning* of the statement is within the (limited) range that the assistant understands, then the system should respond appropriately. Attempts to define specialized English subsets as command languages can be frustrating for users who discover that natural (to them, if not the designer) paraphrases of their requests cannot be understood.

The approach taken in Peedy combines aspects of both knowledge intensive understanding systems and of more pragmatic task-oriented systems. Our system is built on a broad-coverage natural language system which constructs a rich semantic representation of the utterance, which is then mapped directly into a task-based semantic structure. The goal is to provide the flexibility and expressive power of natural language within a limited task domain, and to do so with only a moderate amount of domain-specific implementation effort. In this respect, our approach is most similar to pragmatic natural command language systems, but we have chosen to base our efforts on a rich natural language foundation, so that we will be able to expand the system's linguistic capabilities as the language processing technology continues to develop.

The remainder of section 1.3.1 describes the spoken language processing in the current Persona prototype, focusing especially on the interface between the broad-coverage natural language processing system and the Persona semantic module (labeled NLP and Semantic in Figure 1).

1.3.1.1 Whisper Speech Recognition

Spoken input to the Persona assistant is transcribed by Whisper, a real-time, speaker-independent continuous speech recognition system under development at Microsoft Research [7]. In the current Peedy prototype, all possible user utterances are described to the system by a context free grammar. For example, Figure 3 shows the portion of the grammar which generates the 16 variations of "Play something by madonna after that" that Peedy recognizes.

```
STATEMENT play something by ARTIST TIMEREF
TIMEREF after that
TIMEREF next
ARTIST madonna
ARTIST joe jackson
ARTIST claude debussy
ARTIST andrew lloyd webber
ARTIST synchro system
ARTIST pearl jam
ARTIST joe cocker
ARTIST bonnie raitt
```

Figure 3: Grammar for one legal Peedy statement

The user speaks one statement at a time, using a push-to-talk button to indicate the extent of the utterance. Because Whisper is a continuous recognizer, each sentence can be spoken in a naturally fluid way, without noticeable breaks between words. The recognizer uses a voice model based on speech recorded by a large variety of male English speakers (female speakers use a separate voice model), so no specialized training of the system is required for a new speaker (although the limited grammar currently means that speakers must know which sentences can be understood).

Whisper compares its HMM phoneme models to the acoustic signal, and finds the legal sentence from the grammar that most closely matches input. If the match is reasonably close, it forwards the corresponding text string (along with a confidence measure) to the next module.

1.3.1.2 Name Substitution.

In the music selection task, user utterances may contain the names of artists, songs or albums. These proper names (particularly titles) are likely to confuse a parser because they can contain out of context English phrases: e.g. "Play *before you accuse me* by Clapton". Unfortunately, current speech recognizers cannot detect the prosodic clues that indicate the italics.

```
#1 play track1 by clapton
    track1 = "before you accuse me"
```

```
#2 play before you accuse me by clapton

#3 play track1 by artist1
    track1 = "before you accuse me"
    artist1 = "Eric Clapton"

#4 play before you accuse me by artist1
    artist1 = "Eric Clapton"
```

Figure 4: Possible name substitutions for "Play before you accuse me by Clapton". Alternative #3 is interpreted successfully.

Therefore Peedy includes a name substitution step which scans the input text for possible matches to our database of names and titles (rating them according to plausibility), and substitutes placeholder nouns before passing the input to the parser. Alternative interpretations are presented to the parser (first substituting exact matches, then making no substitutions, and finally trying partial matches), stopping when a successful interpretation is found (Figure 4). Because "Clapton" is only a partial match to the database entry "Eric Clapton", the proper interpretation is not the first one tried, but the earlier ones fail to produce a sensible interpretation.

This approach quite reliably finds the correct interpretation of understandable sentences, but cannot deal with references to names that are not in our database. Currently, such references result in a failure to understand the input.

1.3.1.3 English Parsing.

After names have been substituted, the input string is passed to the MS-NLP English processor, which produces a labeled semantic graph (referred to as the *logical form)* which encodes case frames or thematic roles. For example, the statement "I'd like to hear something composed by Mozart" results in a graph (Figure 5) that represents "I (the speaker) would like that I hear something, where Mozart composed that something." Several strict English paraphrases produce identical logical forms, e.g.:

I'd like to hear something that was composed by Mozart.
I would like to hear something that Mozart composed.
I'd like to hear something Mozart composed.

```
like1 (+Modal +Past +Futr)
 \-Dsub----i1 (+Pers1 +Sing)
  \Dobj----hear1
            \-Dsub----i1
             \Dobj----something1 (+Indef +Exis +Pers3 +Sing)
                       `-Prop----compose1
                                  \-Dsub----mozart1 (+Sing)
                                   \Dobj----something1
```

Figure 5: Logical Form produced by parse of "I'd like to hear something composed by Mozart."

MS-NLP processes each input utterance in three stages:

- *syntactic sketch:* syntactic analysis based on augmented phrase structure grammar rules (bottom-up, with alternatives considered in parallel),
- *reassignment:* resolution of most syntactic ambiguities by using semantic information from on-line dictionary definitions, and

- *logical form:* construction of a semantic graph which represents predicate-argument relations by assigning sentence elements to "deep" cases, or functional roles, including: Dsub (deep subject), Dobj (deep object), Dind (deep indirect object), Prop (modifying clause), etc.

The resulting graph encodes the semantic structure of the English utterance. Each graph node represents the root form of an input word; arcs are labeled by the appropriate deep cases.

1.3.1.4 Application-Specific Transformations

The logical form is then processed by applying a sequence of graph transformations which use knowledge of both the interaction scenario and the task domain. These application-specific transformations recognize:

- artifacts that commonly occur in conversational speech,
- language interpretations that are appropriate in the context of a user-assistant conversation,
- task-specific vocabulary,
- colloquial expressions and specialized grammatical constructs common in the task domain, and
- descriptive qualifications of objects in the application,

and convert them into a normalized domain-specific semantic representation which we call a *task graph* (see Figure 6).

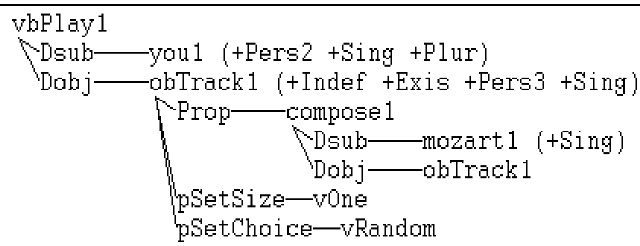

Figure 6: Task graph produced from Figure 5, by application of music assistant transformations.

The task graph represents the same meaning as the logical form, but in terms of the concepts defined within a specific application. The application designer defines:

- *abstract verbs*: which correspond to actions that the assistant can do, or knows about (e.g. vbPlay refers to playing a piece of music),
- *object classes:* which name the categories of conceptual objects in the task domain (e.g. obTrack),
- *object properties:* which label the possible attributes of each object class (e.g. pArtist, pRole), and
- *property values:* which enumerate sets of legal property values (e.g. vComposer, vRandom).

Object properties are used to label arcs in the task graph; the other application identifiers serve as graph nodes.

These application-specific transformations are carried out by rules written in G, a custom language developed as part of the MS-NLP project. Each rule specifies a structural pattern for a semantic graph: whenever the graph for the current utterance matches the pattern, the rule fires. The body of the rule can then modify the semantic graph appropriately.

Our rules are designed to translate a language-based representation of the user's utterance into an unambiguous application-specific representation. The driving force behind these rules is the need to recognize all legitimate English paraphrases of a request and reduce them to a single canonical structure. The canonical form allows the application to deal with a single well-specified representation of meaning, while giving users nearly complete freedom to express that meaning in whatever fashion they find most comfortable.

A single English statement can be paraphrased in a variety of ways: by modifying vocabulary or syntactic structure, or (especially in spoken communication) by employing colloquial, abbreviated, or non-grammatical constructions. In addition, spoken communication occurs within a social context that often alters the literal meaning of a statement. In Persona, we try to identify and deal with each category of paraphrase independently, for two reasons. First, many of our graph transformations might be applicable in related task domains, so they are grouped to facilitate possible reuse. Secondly, our transformations are designed to be applied in combination: each rule deals with a single source of variation and the G processor executes all the rules which match a given utterance. Thus a small collection of individual rules can combine to cover a very wide range of possible paraphrases.

Verbal artifacts: Verbal expression is often padded with extra phrases which contribute nothing essential to the communication (except perhaps time for the speakers to formulate their thoughts). Rules which remove these artifacts, converting (for example):

"Let's see, I think I'd like to hear some Madonna."

into

"I'd like to hear some Madonna."

are appropriate for applications using spoken input.

User-assistant interactions: Persona attempts to simulate an *assistant* helping the user in a particular task domain. This social context evokes a number of specialized language forms which are commonly used in interactions with assistants. For example, polite phrases, such as "please" and "thank you", do not directly affect the meaning of a statement. Other social conventions are critical to a correct understanding of the user's intent; in particular, an expression of desire on the user's behalf should generally be interpreted as a request for action by the assistant. Therefore, Persona includes rules which recognize the semantic graphs for forms such as:

"I'd like to hear some Madonna."

"I want to hear some Madonna."

"It would be nice to hear some Madonna."

and translate them into a graph corresponding to the explicit imperative:

"Let me hear some Madonna."

These transformations would be appropriate for interaction with Persona in any application domain.

Synonym recognition: A major source of variability in English paraphrases comes from simple vocabulary substitution. For each abstract verb and object class in the application, we use a Persona rule to translate any of a set of synonyms into the corresponding abstract term. These synonyms often include ones which are context dependent; for example in our music selection application, "platter" and "collection" are transformed into *obCD*, "music" and "something" become *obTrack*, and "start" and "spin" translate into *vbPlay*. This approach generates correct interpretations of a wide variety of task-specific utterances, including:

"Spin a platter by Dave Brubeck."

"I'd like to hear a piece from the new Mozart collection."

"Start something by Madonna."

However, it does so at the expense of finding valid interpretations for very unlikely statements, e.g.:

"Spin a music from the rock platter."

In practice, we expect this to cause little difficulty within narrow domains; however, as we generalize to related applications, we expect conflicts to arise. By first translating generic or ambiguous words into more general abstract terms (e.g. "Play something" into *vbPlay obPlayable*) we can postpone interpretation to the necessary point, so that in "Play something by Hitchcock", "something" can be resolved as *obMovie* based on the results of the database search.

Colloquialisms: Another class of application-specific transformations deal with specialized grammatical conventions within the domain. To understand a statement like:

"How about some Madonna."

we treat "how about" as equivalent to "play", and employ a rule which recognizes "play artist" as an abbreviation for "play something by artist". In a similar fashion, an isolated object description can be assumed to be a request for the default action, as in: "A little Mozart, please." We expect that each task domain will require a few idiosyncratic rules of this sort, which compensate for the tendency of speakers to omit details which are obvious from the interaction context. In effect, these rules define a model of the default interaction context, which depends only on the task domain. An explicit model of the current dialogue context is used to properly interpret anaphoric references and fragments used to clarify earlier miscommunications (e.g. "The one by Mozart.").

Object descriptions: The majority of our application-specific transformation rules are designed to interpret descriptions of objects within the task domain. Much of the expressive power of natural language comes from the ability to reference objects by describing them, rather than identifying them by unambiguous names. Therefore it is critical that Persona be able to properly interpret a wide variety of domain object descriptions.

A Persona application defines a collection of descriptive properties which can be used to qualify references to objects within the domain. For example, a track from a CD can be described by combinations of the following attributes: title, title of containing CD, position on CD, year, musical genre, energy level, vocal/instrumental, year produced, date acquired, music label, length, or by the names of its singers, composers, lyricists, musicians, producers, etc.

Persona rules evaluate the modifiers of each object in the logical form and transform them into the appropriate property values. Typical examples include:

- adjectives which imply both a property and its value ("jazz CD" implies *pGenre:vJazz*);
- nouns which identify an object and also specify other attributes ("concerto" implies *pGenre: vClassical*);
- cases where the interpretation cannot be determined without additional context ("new CD" could refer to either *pDateAcquired* or *pYearProduced,* so a generic property *pAge* is passed to the action routines); and
- propositional modifiers ("the CD I bought yesterday" transforms into *pDateAcquired: vYesterday).*

While the collection of descriptive attributes will vary for each application, we expect that there will be many similarities across related domains, and it will therefore be possible to migrate many rules into new domains.

1.3.1.5 Action Templates

After all legal transformations have been applied, the resulting task graph is matched against a collection of *action templates* which represent utterances that the application "understands", i.e., knows how to respond to. If the Persona matcher locates a template with the same abstract verb and deep case fillers, then processing continues with the evaluation (e.g. by running a database query) of any *object descriptions* in the task graph. For example, the template for any request that Persona play one or more tracks from a CD:

vbPlay Dsub: you Dobj: obTrack

matches the task graph in Figure 6. Then the description of obTrack, consisting of properties such as pArtist, pSetSize, and pSetChoice can be evaluated. In this case, a database query is executed which finds all tracks in the music collection which have Mozart listed as composer. Finally, an event descriptor corresponding to the matched template (including the results of the object evaluations) is transmitted to the dialogue module.

1.3.2 DIALOGUE MANAGEMENT

Upon receipt of an input event descriptor from the language subsystem, the dialogue manager is responsible for triggering Peedy's reaction: an appropriate set of animations, verbal responses, and application actions, given the current dialogue situation. In the Peedy prototype, that situation is represented in two parts: the current conversational state, and a collection of context variables.

1.3.2.1 Conversational State Machine

The conversational state is represented by a simple finite state machine, which models the sequence of interactions that occur in the conversation. For each conversational state (e.g. Peedy has just suggested a track that the user may wish to hear), the state machine has an action associated with every input event type. The current state machine has just five conversational states and seventeen input events, which results in approximately 100 distinct transitions (in a few cases, there are multiple transitions for a single state/event pair, based on additional context as described below).

Each transition in the state machine can contain commands to trigger animation sequences, generate spoken output, or activate application (CD player) operations. For example, Figure 7 shows the rule that would be activated if Peedy had just said "I have The Bonnie Raitt Collection, would you like to hear something from that?" (stGotCD), and the user responded with "Sure" (evOK). Peedy's response would be to:

- trigger the *pePickTrack* animation, which causes Peedy to look down at the CD (note) that he's holding as if considering a choice,
- expand the description of the current CD into a list of the songs it contains (genTracks),
- select one or two tracks, based on the parameters given in the interaction (doSelect)-- in this example, Peedy would pick one track at random, and
- verbally offer the selected song, e.g. "How about *Angels from Montgomery*?", with the appropriate beak-sync.

State	Event	NewState	Action
stGotCD	evOK	stSuggested	do pePickTrack; genTracks; doSelect; Say <!PDSays(howAbout)>

Figure 7: Example dialogue state transition

1.3.2.2 Context and Anaphora

In addition to the conversational state, the Peedy dialogue manager also maintains a collection of context variables, which it uses to record parameters and object descriptions that may affect Peedy's behavior. This mechanism is used to handle simple forms of anaphora, and to customize behavior based upon the objects referenced in the user's request.

For example, the question "Who wrote that?" generates the action template:

vbTell Dsub: you Dobj: obArtist(pRole: vComposer, pWork: refObX)

which corresponds to the paraphrase "Tell me the artist who composed that work". **refObX** is interpreted as the last referenced object, and the identifier of that object is retrieved from the corresponding context variable. The specified database query is then performed (i.e.: what Artist composed "Angel From Montgomery") and the result is stored in context variables. Then the input event *evWhoWroteTrack* is sent to the dialogue manager. State transition rules can be predicated upon context expressions; so in Figure 8 the appropriate rule will fire, depending on the number of artists that were found by the query, and Peedy will respond with either "Bonnie Raitt" or "I don't know". (More than one artist match isn't currently handled.)

State	Event	Predicate	NewState	Action
stAttending	evWhoWroteTrack	cnt0=1	stAttending	doSelect; Say <!PDSays(artist)>
stAttending	evWhoWroteTrack	cnt0=0	stAttending	Say <!PDSays(DontKnow)>

Figure 8: Dialogue rules for evWhoWroteTrack

1.3.2.3 Verbal Responses by Template Expansion

In the examples above, the *Say* action in a dialogue state transition was used to generate Peedy's spoken output. The argument to *Say* is a template expression, which specifies the category of verbal response that is desired. Figure 9 shows the four templates for the category *haveCD* in the current system, which Peedy would use to respond to "Have you got anything by Bonnie Raitt?" The system selects one of the templates

Category	Prob	Result
haveCD	0.25	i have <!getLastCD><=Title> from <=Year>
haveCD	0.333	ive got <!getLastCD><=Title>
haveCD	0.5	ive got <!getLastCD><=Title>, would you like to hear something from that?
haveCD	1	i have <!getLastCD><=Title> from <=Year>, would you like to hear something from that?

Figure 9: Variations of saying I have a CD

based on the specified probabilities; in this case, the choices are equally likely (the first is chosen 1 in 4 times, otherwise, the second has a 1 in 3 chance, etc.). This allows some variation Peedy's responses, including an occasional cute or silly remark. The selected template is then expanded, by evaluating queries (*getLastCD* loads all attributes of the last referenced CD into context) and substituting context variables (*Title* and *Year* are values assigned by *getLastCD*).

1.3.2.4 Episodic Memory

As illustrated in Figure 10, when Peedy fails to understand a spoken input, he raises his wing to his ear and says "Huh?". This is a natural way to concisely inform users that there was a miscommunication, which quite effectively cues them to repeat. However, when repeated speech recognition failures occur for the same input (as they occasionally do), the exact repetition of the "Huh?" sequence is very awkward and unnatural. This is a basic example of Peedy's need to understand the history of the interaction, and to adapt his behavior accordingly.

We have recently experimented with additions to the prototype system which record a detailed log of events that occur during interactions with Peedy, and then use that history to adjust his behavior to be more natural. The memory has been used to enable three new types of context dependent behavior:

- Depending on previous (or recent) interactions, Peedy's reaction to a given input can vary systematically. For example, the second time he fails to understand an utterance, he says "Sorry, could you repeat that?", and then becomes progressively more apologetic if failures continue to reoccur.
- The selection of an output utterance can depend on how frequently (or recently) that particular alternative has been used. For example, a humorous line can be restricted to be used no more than once (or once a week) per user. (The interaction memory is retained separately for each user.)

- Dialogue sequences can adjust a simple model of Peedy's emotional state (e.g. to be happy because of successful completion of a task, or sad because of repeated misrecognitions). His emotional state can then affect the choice of utterance or animation in a particular situation.

Figure 10: Peedy indicating a misrecognition

1.3.3 VIDEO AND AUDIO OUTPUT

An important element in the "believability" of an agent is its ability to produce richly expressive visual behavior and to synchronize those visual elements with appropriate speech and sound effects. We found that in order to achieve the necessary level of realism and expression, most of the output elements must be carefully authored. The three dimensional model of Peedy's body, his movements, facial expressions, vocalizations and sound effects, were all individually and painstakingly designed. But in order to create a believable conversational interaction, it is equally important that Peedy react quickly and flexibly to what the user says. To make that reactivity possible, we divided the animations and sounds up into short fragments (*authored elements*) and developed a run-time controller for Peedy (called Player) which uses our reactive animation library (ReActor) to sequence and synchronize those elements in real-time. This approach also lets us combine the authored elements into a wide variety of longer animations, so that long repetitive sequences can be avoided.

1.3.3.1 ReActor

ReActor represents a visual scene as a named hierarchy decorated with properties. The hierarchy includes all the visible objects and additional entities such as cameras and lights. Properties such as position or orientation of a camera, the material or color of an object, or the posture of an articulated figure can all be animated over time. Camera (and lighting) control provides the ability to support cinematic camera and editing techniques in a real-time computer graphics environment. More abstract properties of an agent such as its "state of excitement" can also be defined and animated.

ReActor explicitly supports temporal specifications in terms of wall clock time and relative time, where relative time is defined in terms of a hierarchy of embedded time lines. These specifications include when and for how long actions take place. This support for time allows ReActor to also synchronize multiple time-based streams such as sound, speech and animation.

The Scene Hierarchy And Properties: The scene is represented by a *named hierarchy*, which includes all the visible objects and additional entities such as cameras and lights. These are all first class objects which can be manipulated in a uniform way by the animation system.

The hierarchy is decorated with *properties*, which include geometric specifications such as position and orientation. However, as we shall see later, these properties can also be more abstract, where changes are reflected in the visual (or sonic) representation of the object via an application-defined function. Any of these properties can be readily altered, and their changes over time form the basis of all animations.

Properties And Controls: To animate a given property over a specific time interval, a property is bound to a *control.* The control is a function of *wall clock time* which specifies the value of a property. The control may be a standard interpolation function or a more specialized, application-defined function.

Scripts: Scripts specify the bindings of properties to controls during an interval on a local time line. The local time line is translated to wall clock time when the script is invoked. The script is useful for two reasons. First, one can collect related controlled properties into a larger named object which can be invoked as a unit. Second, and more importantly, the script provides a mechanism to describe things in terms of *relative time* rather than wall clock time.

Support For Real-Time: ReActor ensures correct real-time behavior so that events in the underlying model occur at the correct times independent of the rendering process. Relative timings among events are thus always maintained.

ReActor estimates the time at which the next frame will be displayed, and properties are updated to values correct for that time. On a slower (or busier) machine, the update rate will be lower, but the appearance of each frame will be correct for the time at which it is displayed.

ReActor also allows us to specify *critical times*, which are times at which frames must be displayed. Critical times are needed because sometimes a certain instant needs to be portrayed to produce a convincing animation; for example, in a hammering sequence, it is important to show the instant when the hammer hits the nail. At lower frame rates, the use of critical times produces much more satisfying animations.

Similarly we can readily synchronize other types of time-based streams, such as sound. As an example, the sound of the hammer hitting the nail can be made to occur at the time specified for the strike.

Directors: Complex reactive behavior of objects is implemented via *directors*. Our overall goal is to be able to control and animate, in real-time, characters and objects with complex behaviors which respond to user input. Directors, supported by the lower level abstractions, provide this capability. Directors are triggered by various events, including temporal events, changes to properties, user input, and events generated by other directors. Directors create and/or invoke scripts, or directly specify bindings of properties to controls.

In the prototype system, directors are used to give Peedy a variety of subtle ongoing behaviors: he blinks and makes other small movements occasionally, and after a period of inaction will sit down, wave his legs, and eventually fall asleep..

1.3.3.2 Player

ReActor provides tools for scheduling and synchronizing many fine-grained animations. However, the animation requests that are made by Peedy's Dialogue Manager are at a much higher level. These requests trigger fairly long sequences which correspond to complete steps in Peedy's interaction with the user. An animation controller, called Player, is responsible for converting the high-level requests into the appropriate sequences of fine-grained animations. Since the appropriate sequence of scripts to use can depend upon the current state of the character (e.g. standing or sitting, holding a note or not, etc.), selecting and coordinating the scripts to produce natural behavior can involve complex dependencies. Player supports a convenient plan-based specification of animation actions, and compiles this specification into a representation that can be executed efficiently at run time.

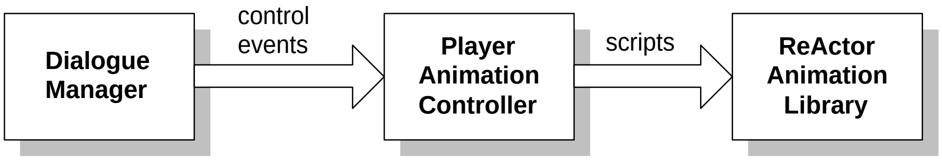

Figure 11: Architecture of Peedy's animation control

Figure 11 illustrates the slice of the Persona architecture that handles animation control. The dialogue manager sends control events to the animation controller. This controller interprets the incoming events according to its current internal state, informs the low level graphics system (ReActor) what animations to perform, and adjusts its own current internal state accordingly.

For example, consider the path of actions when the user asks Peedy "What do you have by Bonnie Raitt?" This is illustrated in Figure 12. First the application interprets the message, and sends a peSearch event to the animation controller, to have Peedy search for the disc. The animation controller knows that Peedy is in his "deep sleep" state, so it sequentially invokes the wakeup, standup, and search animations. It also changes Peedy's current state (as represented in the animation controller) to standing, so that if another peSearch event is received immediately, Peedy will forego the wakeup and standup animations, and immediately perform a search.

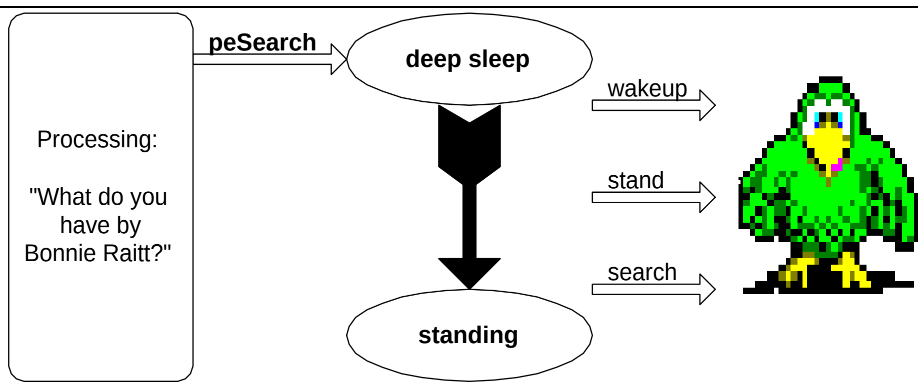

Figure 12: An animation control example

One can view the animation controller as a state machine, that interprets input events in the context of its current state, to produce animation actions and enter a new state. Originally we specified the animation controller procedurally as a state machine, but as new events, actions, and states were added, the controller became unwieldy, and very difficult to modify and debug. It became clear that we needed a different manner of specifying the controller's behavior. One of the difficulties of specifying this behavior is that graphical actions make sense in only limited contexts for either semantic reasons (Peedy cannot sleep and search at the same time) or animation considerations (the search script was authored with the expectation that Peedy would be in a standing position).

Player calculates these transitions automatically, freeing the implementer from part of the chore of constructing animated interfaces. To accomplish this, Player uses planning, a technique traditionally used by the AI community to determine the sequence of operators necessary to get from an initial start state to a goal state. In our system, the operators that affect system state are animation scripts, and the programmer declares preconditions and postconditions that explain how each of the scripts depend on and modify state. One of the major problems with planning algorithms is that they are computationally intensive. Animation controllers, however, have to operate in real time. Our solution is to precompile the conveniently specified planning notation into an efficient to execute state machine.

The language for specifying the behavior of the animation controller has five components. Recall that the animation controller accepts high-level animation events and outputs animation scripts. So the language must contain both event and script definitions. The language also contains constructs for defining state variables that represent animation state, autonomous actions called autoscripts, and a state class hierarchy that makes defining preconditions easier. Each of these language constructs will now be described in turn.

State variables: State variables represent those components of the animation configuration that may need to be considered when determining whether a script can be invoked. State variable definitions take on the form:

(state-variable *name type initial-value* <*values*>)

All expression in the language are LISP s-expressions (thus the parentheses), and bracketed values represent optional parameters. The first three arguments indicate the name, type, and initial value of the variable. State variables can be of type boolean, integer, float, or string. The last argument is an optional list of possible values for the variable. This can turn potentially infinitely-valued types, such as strings, into types that can take on a limited set of values (enumerative types). Examples of state-variable definitions are:

(state-variable ‘holding-note ‘boolean false)

(state-variable ‘posture ‘string ‘stand ‘(fly stand sit))

The first definition creates a variable called holding-note, which is a boolean and has an initial value of false. The second creates a variable called posture, which is a string that is initialized to stand. It can take on only three values (fly, stand, and sit), and this should be expressed to the system because in some cases the system can reason about the value of the variable by knowing what it is not.

There is a special class of state variable, called a time variable. Time variables are set to the last time one of a group of events was processed.

Autoscripts: Autoscripts make it easy to define autonomous actions, which are actions that occur typically continuously when the animation system is in a particular set of states. Examples of this would be having an animated character snore when it is asleep, or swing its legs when it is bored. Autoscripts are procedures that are executed whenever a state variable takes on a particular value. For example, to have the snore procedure called when a variable called alert is set to sleep, we write the following:

(autoscript ‘alert ‘sleep ‘(snore))

The third argument is a list, because we may want to associate multiple autonomous actions with a given state variable value. Note that though we typically bind autoscripts to a single value of a state variable, we could have an autoscript run whenever an arbitrary logical expression of state variables is true, by binding the autoscript to multiple variable values, and evaluating whether the expression is true within the autoscript itself before proceeding with the action.

Event definitions: For every event that might be received by the animation controller, an event definition specifies at a high-level what needs to be accomplished and the desired timing. Event definitions take on the form:

(event *name* <*directives*>*)

The term <directives>* represents a diverse set of statements that can appear in any number and combination. The :state directive tells the controller to perform the sequence of operations necessary to achieve a particular state. The single argument to this directive is a logical expression of state variables, permitting conjunction, disjunction, and negation. This high-level specification declares the desired results, not how to attain these results. In contrast, the :op directive instructs the system to perform the operation specified as its only argument. The animation controller may not be in a state allowing the desired operation to be executed. In this case, the controller will initially perform other operations necessary to attain this state, and then execute the specified operation.

For example, the peBadSpeech event is received by Player whenever our animated agent cannot recognize an utterance with sufficient confidence. Its effect is to have Peedy raise his wing to his ear, and say “Huh?” This event definition is as follows:

(event ‘evBadSpeech :state ‘wing-at-ear :op ‘huh)

When an evBadSpeech event comes over the wire, the controller dispatches animations so that the expression wing-at-ear (a single state variable) is true. It then makes sure that the preconditions of the huh operator are satisfied, and then executes it. Note that wing-at-ear could have been defined as a precondition for the huh operator, and then the :state directive could have been omitted above. However, we chose to specify the behavior this way, because we might want huh to be executed in some cases when wing-at-ear is false.

By default, the directives are achieved sequentially in time. Above, wing-at-ear is made to be true, and immediately afterwards huh is executed. The :label and :time directives allow us to override this behavior, and define more flexible sequencing. The :label directive assigns a name to the moment in time represented by the position in the directives sequence at which it appears. The :time directive adjusts the current time in one of these sequences.

(event ‘evThanks
:op ‘bow
:label ‘a
:time ‘(+ (label a) 3)
:op ‘camgoodbye
:time ‘(+ (label a) 5)
:op ‘sit)

As defined above, when the animation controller receives an evThanks event, Peedy will bow. The label a represents the time immediately after the bow due to its position in the sequence. The first :time directive adjusts the scheduling clock to 3 seconds after the bow completes, and this is the time that camgoodbye operator executes, moving the camera to the “goodbye” position. The second :time directive sets the scheduling clock to 5 seconds after the bow, and then Peedy sits. If Peedy must perform an initial sequence of actions to satisfy the sit precondition, these will begin at the this time, and the sit operation will occur later. Note that these two timing directives allow operations to be scheduled in parallel or sequentially.

Four additional directives are used, albeit less frequently. The :if statement allows a block of other directives to be executed only if a logical expression is true. This allows us, for example, to branch and select very different animation goals based on the current state. Occasionally it is easier to specify a set of actions in terms of a state machine, rather than as a plan. The :add and :sub directives change the values of state variables, and in conjunction with the :if directive, allow small state machines to be incorporated in the controller code. The :code directive allows arbitrary C++ code to be embedded in the controller program.

Operator definitions: Scripts are the operators that act on our graphical scene, often changing the scene’s state in the process. Operator definitions are of the following form:

(op *opname* <:script *scriptname*>
<:precond *precondition*>
<:add *postcondition*>

<:sub *postcondition*>

<:must-ask *boolean*>)

This creates an operator named opname associated with the script called scriptname. The operator can only execute when the specified precondition is true, and the postcondition is typically specified relative to this precondition using :add or :sub. Since operators typically change only a few aspects of the state, relative specification is usually easiest. The :must-ask directive defaults to false, indicating that the planner is free to use the operator during the planning process. When :must-ask is true, the operator will only be used if explicitly requested in the :op directive of an event definition. An example script definition appears below:

```
(op     'search
        :script 'stream
                :precond '((not holding-note) and ...)
        :add 'holding-note)
```

This defines an operator named search, associated with a script called stream. The precondition is a complex logical expression that the state class hierarchy, described in the next section, helps to simplify. The part shown here says that Peedy cannot be holding a note before executing a search. After executing the search, all of the preconditions will still hold, except holding-note will be true.

Though we have so far referred to operators and scripts interchangeably, there are really several different types of operators in Player. Operators can be static scripts, dynamic scripts (procedures that execute scripts), or arbitrary code. In the latter two cases, the :director or :code directives replace the :script directive.

We can also define macro-operators, which are sequences of operators that together modify the system state. As an example, the hard-wake macro-operator appears below:

```
(macro-op       'hard-wake
        :precond '(alert.snore and ...)
        :add 'alert.awake
        :seq '(:op snort :op exhale :op focus))
```

The above expression defines a macro-operator that can only be executed when, among other things, the value of alert is snore. Here, the '.' ("dot") comparator denotes equality. Afterwards, the value of alert will be awake. The effect of invoking this macro-operator is equivalent to executing the snort, exhale, and focus operators in sequence, making Peedy snort, exhale, then focus at the camera in transitioning from a snoring sleep to wakefulness in our application. The :time and :label directives can also appear in a macro definition to control the relative start times of the operators, however, our system requires that care be taken to avoid scheduling interfering operators concurrently.

State class hierarchy: In the last two examples, the preconditions were too complex to fit on a single line, so parts were omitted. Writing preconditions can be a slow, tedious process, especially in the presence of many interdependent state variables. To simplify the task, we allow programmers to create a state class hierarchy to be used in specifying preconditions. For example the complete precondition for the search operator defined earlier is:

```
((not holding-note) and alert.awake and
posture.stand and (not wing-to-ear) and
(not wearing-phones))
```

Since this precondition is shared by five different operators, we defined a state class (called standing-noteless) that represents the expression, and is used as the precondition for these operators. This makes the initial specification easier, but also subsequent modification, since changes can be made in a single place.

Class definitions take the following form:

(state-class classname states)

State class hierarchies support multiple inheritance. Here, states is a list of state variable expressions or previously defined state classes. A state-class typically inherits from all of these states, and in the case of conflicts, the latter states take precedence. State hierarchies can be arbitrarily deep. The stand-noteless class is not actually defined as the complex expression presented earlier, but as:

(state-class stand-noteless
'(stand-op (not holding-note)))

In other words, the stand-noteless class inherits from another class called stand-op. We have found that the semantics of an application and its animations tend to reveal a natural class hierarchy. For example, for our animated character to respond with an action, he must be awake, and for him to acknowledge the user with an action, he must not have his wing to his ear as if he could not hear, and cannot be wearing headphones. These three requirements comprise the class ack-op (for acknowledgment operation), from which most of our operations inherit, at least indirectly.

Algorithm: Typical planning algorithms take a start state, goal state, and set of operators, compute for a while, then return a sequence of operators that transforms the start into the goal. Since our animated interface must exhibit real-time performance, planning at run-time is not an option. Instead, Player pre-compiles the plan-based specification into a state machine that has much better performance. This places an unusual requirement on the planning algorithm—it must find paths from any state in which the system might be to every specified goal state.

A naive approach might apply a conventional planner to each of these states and goals independently. Fortunately, there is coherence in the problem space that a simple variation of a traditional planning algorithm allows us to exploit. Our planning algorithm, like other goal regression planners, works by beginning with goals and applying operator inverses until finding the desired start state (or in our case, start states). The algorithm is a breadth-first planner, and is guaranteed to find the shortest sequence of operators that takes any possible start state to a desired goal.

The next step, after the planning algorithm finishes, is to build the actual state machine. Our system generates C++ code for the state machine, which is compiled and linked together with the Reactor animation library and various support routines. The heart of the state machine has already been calculated by the planner. Recall that plans are (state conditional, action sequence) pairs, which the planner computed for every goal state. These plans can readily be converted to if-then-else blocks, which are encapsulated into a procedure for their corresponding goal. These procedures also return a value indicating whether or not the goal state can be achieved. We refer to these procedures as state-achieving procedures, since they convert the existing state to a desired state.

Next, the system outputs operator-execution procedures for every operator referenced in event definitions. These procedures first call a state-achieving procedure, attempting to establish their precondition. If successful, the operator-execution procedures execute the operator and adjust state variables to reflect the postcondition. When multiple operators share the same precondition, their operator-execution procedures will call the same state-achieving procedures.

Finally, we generate event procedures for every event definition. These procedures, called whenever a new event is received from the application interface, invoke state-achieving procedures for each :state directive, and operator-execution procedures for each :op directive in the event definition. The :time directive produces code that manipulates a global variable, used as the start time for operator dispatch. The :label directive generates code to store the current value of this variable in an array, alongside other saved time values.

The planner and ancillary code for producing the state machine are implemented in Lucid Common Lisp, and run on a Sun Sparcstation. Our animation controller specification for the

Peedy prototype contains 7 state variables (including 1 time variable), 5 auto-scripts, 32 operators, 9 state classes, and 24 event definitions. The system took about 4 seconds to generate a state machine from this controller specification on a Sparcstation 1+, a 15.6 MIPS, 1989-class workstation.

It is important to note that in our Peedy application, not all animation is scheduled via planning. We have found that low-level animation actions, such as flying or blinking, are conveniently implemented as small procedural entities or state machines that are invoked by the higher-level animation planner. These state machines can be activated through autoscripts and the :director directive, and they can maintain their own internal state, or reference and modify the animation controller's state variables at run-time. As mentioned earlier, state machines can also be embedded into the animation controller using the event definition's :if directive. Our experience suggests that planning-based specification should not entirely replace procedurally based specification. The two techniques can best be used together.

1.3.3.3 Speech and Sound Effects

In the audio component of Persona, we set out to give Peedy an appropriate voice, and to place him in a convincing aural environment. Our goals include the ability to easily add new remarks to the character's speech repertoire, and to synchronize the audio properly with his lip (or beak) movement. Because speech and sound effects have such a large effect on the user's perception of the system, we think it's important to concentrate significant effort on attaining aural fidelity and richness-- both by situating cinema-quality sound effects properly within a realistic acoustic environment, and by maximizing the naturalness and emotional expressivity of the character's voice.

The character's voice needs to sound natural while having a large vocabulary. Text to Speech (TTS) systems can deliver excellent language coverage but the quality of even the best TTS products destroys the anthropomorphic illusion of the agent. In the prototype, we chose instead to pre-record speech fragments; which vary from single words ("one", "Madonna") to entire utterances ("Another day, another CD. What do you want to hear?").

To maintain a suspension of disbelief it is critical that Peedy's voice be synchronized with his visual rendering. To get accurate "beak sync", we analyze each speech fragment with the speech recognition system to determine the offset of every phoneme within the recording. This information is then used to automatically create a ReActor script which plays the audio fragment and synchronizes Peedy's beak position to it. (Sound effects are handled similarly, except that they are triggered by commands placed into animation scripts by hand.) When the Dialogue Manager selects a statement for Peedy to say, it is broken up into its predefined fragments, and a sequence of corresponding script activations is sent to ReActor.

This approach means that every phrase that the character uses must be individually recorded, a tedious process which makes additions to Peedy's vocabulary difficult. The current system is also limited to producing one sound effect or vocalization at a time, which limits the richness possible in the soundscape. In addition, application actions (e.g. control of CD audio) are currently not triggered by the animation system and are therefore difficult to synchronize properly.

1.4 FUTURE DIRECTIONS

As we had hoped, the creation of a lifelike conversational assistant has proven to be a powerful force for discovering and exploring interactions among a wide variety of efforts at Microsoft Research. Many significant research challenges remain before the creation of a competent "assistant" will become feasible. We list here research topics in which we have active efforts that we feel are critical to continued progress toward that goal.

1.4.1 Language

The speech recognizer in the prototype will only understand sentences which appear in its grammar; however, writing a grammar for all likely utterances (even about a limited domain like music selection) is very difficult. Instead, we would like to switch to an approach which uses a statistical grammar, so that the recognizer searches for matches to the acoustic data where each sequence of words within the match occurs frequently in common speech. Developing a stochastic grammar for conversational speech (including common disfluencies) is therefore an important research objective.

While we feel that our approach to collapsing paraphrases into canonical utterances by means of application-specific transformations is promising, the specification of those transformations is currently too difficult. We are exploring the creation of tools that let application developers define those rules simply by providing examples of the canonical statement and of paraphrases which should be treated equivalently.

1.4.2 Dialogue

We've found that the current dialogue manager based on a simple finite state machine doesn't give us enough flexibility. We're working to reorganize it as a collection of rules which make it easier to handle sub-dialogues, multiple active goals, and character initiation of interchanges. We'd also like to explore ways in which Bayesian decision theory might be used to control the character's responses to events.

Our experiments with giving Peedy an episodic memory and simple emotional response to his interactions with the user have convinced us that those capabilities can give him an important additional sense of naturalness and sociability. We plan to include and extend them in future versions of the system.

We expect that Peedy's abilities will remain limited to quite narrow task domains for the foreseeable future. However, we think it may be feasible for Peedy to have enough background knowledge to guide a new user into his area of competence through a natural conversation. This would involve a mixture of knowing about things that new users are likely to say and having strategies for dealing constructively with input that lies completely beyond his range of understanding.

1.4.3 Video and Audio Output

For the creation of more realistic and variable animations, we plan to focus on the use of ReActor directors to control subtle behaviors. For example, a director can be used to create intelligent cameras which track moving objects automatically, or to adjust the parameters of animations and sound effects to reflect Peedy's emotional state.

The addition of inverse kinematics to the ReActor runtime system is another goal. This capability will allow the animator to author natural motions for just a few components of a complex linked figure (e.g. hands and feet) and let the system calculate appropriate motions of the rest of the figure. This has the potential to significantly reduce the effort required to author new animations.

We are also investigating improvements to the quality of speech synthesis systems by using rules based on a deep language analysis of the input text. That analysis might allow us to automatically generate a natural rhythm and pitch contour for our character's speech, and free us from the need to prerecord all spoken output.

1.5 REFERENCES

1. Badler, N., Phillips, C.B. and Webber, B.L. *Simulating Humans: Computer Graphics, Animation, and Control.* Oxford University Press, 1993.

2. Ball, J.E., *et al.* Demonstration of ReActor: A System for Real-Time Reactive Animations. In *CHI'94 Conference Companion* (April 24-28, Boston, MA), ACM/SIGCHI, N.Y., 1994, pp. 39-40.
3. Bates, J. The Role of Emotion in Believable Agents. *Communications of the ACM.* 37, 7 (1994) 122-125.
4. Cahn, J.E. Generating Expression in Synthesized Speech. B.A., Mills College, Oakland, CA, 1983.
5. Clark, H.H. *Arenas of Language Use.* The University of Chicago Press & The Center for the Study of Language and Information, Chicago, 1992.
6. Cohen, P.R. The Role of Natural Language in a Multimodal Interface. In *ACM Symposium on User Interface Software and Technology* (November 15-18, Monterey, CA), ACM/SIGGRAPH/SIGCHI, N.Y., 1992, pp. 143-149.
7. Gaver, W.W., Smith, R.B. and O'Shea, T. Effective Sounds in Complex Systems: The Arkola Simulation. In *Proceedings of CHI '91: Human Factors In Computing Systems* (April 27-May 2, New Orleans, LA), ACM/SIGCHI, N.Y., 1991, pp. 85-90.
8. Goldberg, A. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA, 1984.
9. Hayes-Roth, B., Sinkoff, E, Brownston, L., Huard, R. and Lent, B.. Directed Improvisation with Animated Puppets. In *CHI '95 Conference Companion* (May 7-11, Denver, CO), ACM/SIGCHI, 1995, pp. 79-80.
10. Hodgins, J.K. Simulation of Human Running. In *Proceedings of the IEEE International Conference on Robotics and Automation* IEEE, 1994, pp.
11. X. Huang, A. Acero, F. Alleva, M. Hwang, L. Jiang, and M. Mahajan: Microsoft Highly Intelligent Speech Recognizer - Whisper", International Conference on Acoustic, Speech, and Signal Processing, 1995, Detroit, USA.
12. Jensen, K., Heidorn, G.E. and Richardson, S.D. (eds.). *Natural Language Processing: The PLNLP Approach.* Kluwer Academic Publishers, Boston, MA, 1993.
13. Lewis, J.B., Koved, L. and Ling, D.T. Dialogue Structures for Virtual Worlds. In *Proceedings of CHI '91: Human Factors in Computing Systems* (April 27 - May 2, New Orleans, LA), ACM/SIGCHI, 1991, pp. 131-136.
14. Maes, P. Agents that Reduce Work and Information Overload. *Communications of the ACM.* 37, 7 (1994) 31-40.
15. Microsoft. *Microsoft Bob*. Redmond, WA, 1995.
16. Nass, C.S., Jonathan; Tauber, Ellen R. Computers are Social Actors. In *Proceedings of CHI'94: Human Factors in Computing Systems* (April 24-28, Boston, MA), Association for Computing Machinery, 1994, pp. 72-77.
17. Nielson, J. Noncommand User Interfaces. *Communications of the ACM.* 36, 4 (1993) 83-99.
18. Norman, D.A. *The Design of Everyday Things.* Doubleday, New York, NY, 1988.
19. Perlin, K. A Remarkably Lifelike Implementation of a Synthetic Computer Character. In *Lifelike Computer Characters '94* (October 4-7, 1994, Snowbird, UT), 1994, pp. 73-74.
20. Schoppers, M.J. Universal Plans for Reactive Robots in Unpredictable Environments. In *IJCAI '87 Conference Proceedings* (August 23-28, Milan, Italy), Morgan Kaufmann, 1887, pp. 1039-1046.
21. Smith, C., Irby, C., Kimball, R., Verplank, B. and Harslem, E. Designing the Star user interface. *Byte.* 7, 4 (1982) 242-282.
22. Takeuchi, A. and Nagao, K. Communicative Facial Displays as a New Conversational Modality. In *INTERCHI'93 Conference Proceedings* (April 24-29, 1993, Amsterdam, The Netherlands), ACM, 1993, pp. 187-193.

23. Vere, S. and Bickmore, T. A Basic Agent. *Computational Intelligence.* 6, 1 (1990) 41-60.
24. Walker, M.A. and Whittaker, S. An investigation into discourse segmentation. In *Proceedings of the 28th Annual Meeting of the ACL* ACL, 1990, pp. 70-79.
25. Ward, W. Understanding Spontaneous Speech: The Phoenix System. In *Proceedings of 1991 International Conference on Acoustics, Speech, and Signal Processing* (May 14-17, Toronto, Canada), IEEE Signal Processing Society, 1991, pp. 365-367.
26. Waters, K. A Muscle Model for Animating Three-Dimensional Facial Expressions. *Computer Graphics (SIGGRAPH'87).* 21, 4 (1987) 17-24.