Adding Backchannel and Turn-Taking Behavior to a Typing Interface

by

Adrian Banard

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 23, 1997

Author .....
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by .....
Justine Cassell
AT&T Career Development Assistant Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Department of Electrical Engineering and Computer Science

Adding Backchannel and Turn-Taking Behavior to a Typing Interface

by

Adrian Banard

Submitted to the Department of Electrical Engineering and Computer Science, on May 23, 1997, in Partial Fulfillment of the Requirements for the Degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

A simple backchannel and turn-taking system was added to a storytelling agent. This project improved upon a typical typed-input interface by incorporating ideas from discourse theory. It sought to create meaningful backchannel and turn-taking behavior while working within the problematical limitations of the usual computer interface consisting of a monitor and keyboard. The storytelling agent gained a responsive and engaging set of behavior, altering the paradigm and feel of a typing interface.

Thesis Supervisor: Justine Cassell
Title: AT&T Career Development Assistant Professor

Table of Contents

Introduction

This project is a modification of the typing input system for SAGE, Marina Umaschi's project, "Soft Interfaces for Interactive Storytelling: Learning about Identity and Communication" (Umaschi, 1997). SAGE is a storytelling agent program which is graphically programmable. Users create and program characters in SAGE which can converse and tell stories. The new interface uses cues such as pause length and conjunctions to determine feedback and turn-taking behavior. It brings the person's interaction with SAGE "closer" (though it is still textual) to human-human interaction, making the interface more usable.

Background

One of the current challenging problems in designing computer interfaces is allowing people to use the skills they have developed in interpersonal discourse. The original computer interfaces were not designed to be comfortable for people, but simply to make the computers work. Recently, much more attention has been paid to trying to make the interface communicate in a way which people find more natural.

Computer users are generally quick to anthropomorphize computers and expect discursive interaction with them, even in obviously artificial situations. In a serious of studies conducted by Reeves and Nass, users have ascribed gender or submissiveness (among other qualities) to computers even when given only minimal cues (Moon, 1997, and Nass, 1997).

To capitalize on the anthropomorphizing expectations of users, various attempts have been made to introduce personae into the computer interface. By persona, I mean the characterization of a computer program as having a personality. Programs talk, have faces, and attempt a general simulation of personality. Much of the work on improving these personae tries to make them more "intelligent" or personable. Other work focuses on the inter-

face, trying to create a computer which attains its personality by behaving as a person would behave.

Creating a more discursive and comfortable interface is very important for programs with personae, but also for any program that tries to interact with a computer user. Bringing the human-computer interface closer to interpersonal discourse will help satisfy user's expectations of the computer.

Recent research in the Media Lab has focused on multimodal interfaces, such as the Gandalf project (Thorisson, 1996). While these multimodal interfaces are very likely the future of computer interface, the current level of speech recognition systems (as well as vision systems and robotics) tends to make this sort of project very difficult. No current speech recognition system can provide a topic-independent, speaker-independent grammar, and they also tend to fail to recognize accented language or the speech of children. (Markowitz, 1996) While multimodal projects such as Gandalf are being completed and speech recognition systems are being improved, the question of what can be done to improve the old paradigm of the keyboard and monitor interface becomes important.

Agent programs often have text-only interfaces, avoiding the technical problems of a multimodal interface. Julia (Maudlin, 1994), one the most famous Chatterbots, was built in a MUD environment. In a MUD Julia can imitate a user fairly well, but only because the environment is primarily limited to text messaging and text-based simulated location and movement. If Julia were in an environment where visual and verbal cues were present, imitating a user would be much more difficult. The Turing Test (Maudlin, 1994), as embodied in the Loebner Prize, similarly tests a program's ability to operate in a testual environment only. If agents are going to be believable characters, their interfaces must be improved, both by adding modalities of communication and by improving the emulation of human behavior in the interfaces.

Textual Computer Interface

In the era of the technological paradigm, the favored computer interface is text-in (through the keyboard) and text-out (through characters printed on the monitor). This has not changed since the early days of the computer use, though personal interaction with the computer has become much more varied and rich. The interface has adapted in some ways, such as through the creation of pointer systems like the mouse. However, the basic paradigm of the computer interface remains a dyadic messaging system. The user types in a message and then hits the return key to signal that the message is complete. The packaged message is then "sent" to the computer system, which processed it and sends a message back to the user, in the form of displayed text. Sometimes the user messaging is done with mouse events instead of keyboard input, or the computer messaging is done using synthesized speech or animations, but the dyadic messaging (or input/output) paradigm still applies.

Unfortunately, this paradigm does not simulate interpersonal communication well. First, talking between people is often not dyadic. Lecture situations, overhearers, and multiple-person conversation all involve more than two people, and are important as discourse situation (Goffman, 1983). Second, conversation is much more complex than discrete back-and-forth messaging. All or most of the participants in a given interaction provide feedback even when they are not speaking, in the form of side utterances, gaze, gesture, or body stance (among other modalities) (Cassell et al, 1994). Specifically, backchannel is the term given to a listener's habit of using gaze, facial expression, short words, and other methods to indicate attention or inattention, or to attempt to release or take the turn. (Schegloff, 1982) Turn-taking is the process of negotiating who is the speaker and who is the listener. (Goodwin, 1981, and Sacks et al, 1974)

Computers fail to provide this feedback while the user is entering text, and computers fail to interpret user feedback while it is the computer's "turn" to speak. Both of these are problems which need to be addressed in basic computer interface, but this thesis will focus on the first, namely the question of producing adequate feedback for a computer user who is attempting to communicate with the computer.

A number of agent programs, such as ELIZA (Weizenbaum, 1966) and SAGE (Umaschi. 1996) use typing as their input form. ("Agent" is being used here in the sense of any program which takes on a character or persona, but see Foner, 1993.)As a messaging system, this input form does not help the believability of the agent. When the program stops talking or printing, it halts and waits for the person to write a full piece and press the enter key. It will wait forever (and often does wait for a very long time) for the user to respond. Unlike a person, it will not get bored, or make small sounds to indicate that it is waiting, or walk away. Similarly, it will not encourage the typist or notice when they have provided cues that they are finished (aside from the return key). This is especially important for SAGE, as it was designed to be used by children as an educational tool. I have seen children stop typing at SAGE and then sit back without pressing return, expecting the computer to interpret their actions as finishing their turn. When these expectations are disappointed, the believability of the computer character is hurt. By "believability", I mean the degree to which users can pretend they are interacting with a person instead of a program. When a computer system is not believable enough, users stop interacting with it like a person and treat it like a computer, which tends to remove the advantages a program gets by taking on a persona.

Improving such typing interfaces by satisfying the expectations of users is not an easy or one-time task, however. First, it is hard to predict how the user's experience conversing with other people will influence their expectations of the computer's actions during typing,

as typing is very different from speaking. Users do have some expectations, but determining which backchannel and turn-taking behaviors map from speech to typing is an open question. For example, will children using a computer expect it to be able to interpret their pauses? These expectations will also differ depending on how much familiarity a person has with a typing interface. Experienced users of computers have become familiar with the typical typing interface of a computer, so they are less likely to expect it to exhibit human-like conversation dynamics.

Second, conventions of backchannel and turn-taking vary widely across cultures and even across parts of the same culture, including variations in pause length, word frequency and choice, and responsiveness, among other things. So when designing some sort of behavior for an agent or program, the specific situation may change the required behavior drastically, depending both on who is using the program and what sort of persona the program is emulating. For instance, the SAGE program assumes the personae of various cultural storyteller figures (Umaschi, 1997), and it should be expected that each storyteller should have its own feedback and turn-taking behavior. Similarly, it seems reasonable that people from different cultures or subcultures or of different ages would expect different backchannel behaviors from a computer. Any system attempting to provide this feedback will have to be flexible or adaptive enough to accommodate these different styles.

There are situations where it is possible to study backchannel in all-text formats. A prime example is the talk program on most UNIX systems, which allows two people to type simultaneously into two parts of a terminal as a method of communication. Each user sees the other's keystrokes as they occur. At any point, either person can (and often does) leave the terminal, which is only indicated by the absence of keystrokes. Thus, to indicate that they are still present, people tend to insert dots or short words, take quick turns, or type simultaneously. In this way a form of backchannel is maintained. Another type of

backchannel which occurs in the talk program is the simultaneous query. One person will ask a short question (such as "How?") while the other is in mid-statement. Similarly, turn-taking is often negotiated by simply stopping typing, hoping the other person picks up the flow, and querying if the other person does not. Using talk demands a certain attention, as it is considered rude to do something else while the other person is on the line. There has been little or no research on programs such as talk. Cherny describes talk as a "simultaneous", "two-way" interaction in the introduction to her dissertation on MUD interaction, but she does not closely examine backchannel and turn-taking behavior with the program (Cherny, 1995). While many of the directions in this project will be motivated by personal experience with talk, an in-depth study of the program and the styles users take with it should be undertaken to better understand user expectations.

Cherny has also done significant research on backchannel and turn-taking strategies on MUDs (Cherny, 1995). MUDs are multi-user text messaging chat spaces. It is not possible to take the turn from another person on a MUD, as users enter messages simultaneously. But some turn-taking behavior has been developed for MUDs. This usually takes the form of other modalities expressed through text. Characters can nod, direct their comments (replacing some of the function of gaze in face-to-face conversation), and express paraverbals. Users also tend to restrict their statements to a certain length, providing space for backchannel or interruption. Similarly, frequent short backchannel actions or comments are used to indicate attention, comment on what is being said, or negotiate turn-taking. Most of this backchannel is in the form of simulated action, but some backchannel is the textual equivalent of verbal backchannel. Cherny also comes to the conclusion that face-to-face backchannel and turn-taking behavior do not map well to MUD situations, due to the lack of simultaneous composition and reception of communication.

A Flexible Backchannel and Turn-Taking System

The purpose of this project was to try to improve a typical typing interface by adding backchannel and turn-taking behavior, based on discourse theory and type of interaction characterized by the talk program. It operated within the (admittedly problematical) limitations of a typical computer interface consisting of a monitor and a keyboard. The backchannel project was therefore a mix of a non-speaking interface (typing) and discourse phenomena from speaking situations.

The project was limited to the user's typing time, and did not address the other half of the backchannel problem (namely having the computer respond to the backchannel behavior of the person sitting in front of it). This other half would require that the computer be able to detect the user's gaze, discussion with other people, and gesture, among other things, and these sensors are not found in a typical computer setup. Agents such as Gandalf are currently developing such backchannel response. Dealing with user interruption (when the person starts typing while the computer is still talking or outputting) is its own larger problem. It should be dealt with very differently by various agents and various non-persona systems (see Klein, 1996, for one example), so this project will not include user-initiated interruption in its domain.

Background Work

A number of backchannel systems have already been produced. Gandalf (Thorisson, 1996), uses facial expression, gesture, body stance, and other modes of feedback to indicate attention and to guide turn-taking. Gandalf also tries to interpret similar cues given by the person conversing with it. Cassell's (Cassell, 1994) Animated Conversation project uses facial expression and gestural backchannel. In systems such as these, backchannel is expected and useful, and the interface is therefore much more believable (Thorisson and Cassell, 1996). Julia has some programming regarding when to speak and when to listen

(in order to survive in MUDs, where users can speak at any time), but she does not negotiate turn-taking or provide explicit backchannel (Maudlin, 1994). In general, backchannel and cued turn-taking has not been implemented in text-to-text interfaces or agents.

SAGE (Umaschi, 1997) stands for Storytelling Agent Generation Environment. Children use SAGE to create storytelling agents which they can then interact with. These agents follow a conversation script, asking the user various questions and then telling them an appropriate story based on their answers. As these agents are produced by various children, they take on personae from a wide variety of situations. A couple of examples are a Rabbi, a basketball player, and a French grandmother.

SAGE undertakes a variety of educational tasks. When children create their own SAGE characters, they get a strong feel for computer programming. Working on a detailed description of the interaction (and testing the results) gives the kids an understanding of the complexity involved in everyday communication. Also, the children assemble the narratives for the storytellers, an exercise which prompts them to research the background and culture of the persona.

Children also explore their own identity through SAGE. First, when they interact with a SAGE character, the persona typically asks them to describe a problem or story in their lives, and then tries to relate their input to a story, asking the children to reflect on the connection between the story and their life. Second, kids tend to pick sage figures that they are familiar with when creating their own SAGE avatar, giving them a chance to explore their own social circle and identity.

While a given agent is operating, input is from the keyboard, and output is either textual or text combined with speech (either synthesized or recorded). Until now, the program simply freezes while the person is typing, not offering feedback until after the user presses

the return key. Comic-style text "balloons" are used to identify the text produced by the SAGE persona and by the user.

A SAGE persona is embodied in a stuffed animal, which complicates and enlivens the output. While output is always textual, speech output can be piped out through the stuffed animal, and it can express certain body movements as well.

The backchannel and turn-taking system was built as an addition to the SAGE program. Unlike other facets of SAGE, it is not modifiable from within the graphical program, because we would need a better understanding of simple ways to parameterize and explain the backchannel process before children could alter it easily. Instead, a set of backchannel behavior is loaded in with each SAGE character, and that character will exhibit the behavior.

Capabilities of the Program

One of the basic tasks of the backchannel system is to provide textual cues which show that the computer is listening. The agent says short words such as "yes", "okay", or "uh huh" while the user is typing, preferably during appropriate pauses. There are other textual cues which the system can provide as well, most notably ellipsis (a series of periods).

The program also provides some feedback which is specific to what the person has already typed in. A high-level version of this would detect information that the user has entered, and ask appropriate questions or make appropriate comments. However, due to the limitations of current text parsing and the relatively small computation time available to the system, it can only be able to make obvious comments, such as rephrasing the last word of the input as a question when appropriate. For instance, if the typist has just entered "and", the system can ask "And?".

When users do not start their turn at an appropriate time, the backchannel system prompts to make it obvious to the person sitting in front of the computer that it is his/her turn to communicate. Similarly, the character provides appropriate hints or sayings when the user pauses for too long, or interprets this pause as the user finishing.

The second primary goal of the backchannel project is to pick up enough cues from the typist to determine when they have decided to end their turn and let the SAGE character continue speaking. The traditional user-computer cue (the return key) still functions, because people are often used to it. The persona also ends turns based on significant pauses in typing. (What qualifies as an appropriate pause is problematic, as people use a variety of cues to negotiate turn-taking. The system just uses a slightly longer than normal pause.) There are cues available in textual input that are not available in speaking, most obviously punctuation. The turn-taking system includes punctuation in its decision to interrupt or not. For instance, it decides to not end the turn if the user last entered a comma, implying they wish to continue.

Another way the user might indicate continuation is through the use of certain cue words. Specifically, words such as "and" and "but" are not intentionally placed at the end of a turn, and evince a wish to continue (Schiffrin, 1987). When the user has recently entered one of these cue words, the system knows that they wish to continue, and does not interrupt their turn, perhaps prompting them instead.

Technical Design of System

The operation of the backchannel system is controlled by an event-based rule system. There are certain events, or outputs, created by the system, primarily outputting textual backchannel and ending the user's turn. Similarly, the backchannel and turn-taking pro-

gram are able to look at various parameters such as timing or word choice to determine when these events should occur.

Each SAGE character can load a load a file containing a set of backchannel rules. These rules determine the backchannel behavior while that character is running. Each rule is an action and a set of requirements. The action is performed when the set of requirements is satisfied. The rules are logically complete, so any logical combination of requirements can be used to satisfy an event.

There are two main advantages to this event/requirement structure. First, it is easy to modify the backchannel rules for a given SAGE character, changing or adapting its behavior with only a small bit of editing. Second, instead of growing as rules are added (there are a very large number of possible backchannel rules), the system grows as requirements or event types are added, and it is relatively easy to add these units. (It can be done simply by implementing the appropriate check or functionality).

Actions Taken by the System

Currently there are four actions that the system can perform. The first is to say one of a set of words or phrases. (Which word or phrase is said is randomly determined.) There can be any number of these sets of words, each with its own set of requirements, and they can interleave or otherwise combine seamlessly. Specifically, it is possible to ensure that the actions do not occur to close together, avoiding collisions where two feedback words or phrases are given at almost the same time.

The second action is rephrasing the last word as a question. This is a conveniently easy backchannel comment that reflects the user's own words back, essentially asking them to clarify or continue. So, for instance, if the person says, "I am going to the store", the system could ask "Store?" While this sounds simple, it is not far from the sort of one-word

question feedback we give each other, such as "Really?" However, determining out when it makes sense for the backchannel system to reflect words in this manner is difficult, so the current system only reflects certain cue words, such as "because".

Ending the user's turn is the third type of action. The user loses the ability to type into the window, and the SAGE character starts speaking. While this is a simple action to take, determining when it should happen is very difficult.

The fourth action is ending the user's control. By control, I mean the person who is leading the conversation (Whittaker and Stenton, 1988). The person "in control" typically initiates a section of dialogue, and holds a position of activity as long as the section continues, no matter who is talking. Control is associated with a given topic. So, for example, if one person asks a question about a particular subject, they would be considered to have control while the subject is discussed. While ending control (as opposed to ending turn) is was considered for the backchannel system, SAGE is currently unable to utilize it, because the user never takes strong control of the conversation during most SAGE interactions. This is because SAGE typically initiates sections of dialogue, to focus the conversation into a tight domain, making it easier for the program to interpret input. Computer-controlled initiation is very common in computer interfaces.

Parameters Observed by the System

One of the most important parameters which the backchannel and turn-taking program monitors is timing. Currently, timing requirements can check against three different times: since the last keystroke, since the beginning of the user's turn, and since the last verbal/textual feedback given by the backchannel system. The first timing variable is used to determine pause length. The second facilitates interactions during the beginning of the user's turn (especially when they are not typing) and can be used to limit the total turn

length. The third parameter (which is effectively the time between feedback comments) is present so the backchannel program can avoid feedback collisions, and space feedback appropriately.

SAGE often asks very direct questions, and is looking for certain things in the answer. The backchannel and turn-taking module also checks for a sufficient input while the user is typing, by simply calling the same functions SAGE calls. Thus, one parameter of the system is whether or not the SAGE character considers the user's turn to be sufficient. This sufficiency or lack of sufficiency is used to alter the likelihood of the character ending the user's turn, for example. For example, if SAGE is looking for a name, then, the input is considered sufficient if it could be a name, or a sentence containing a name.

The backchannel program also checks for words anywhere in the current typing string, or check the last word. The computer determines if the user has just entered a cue word that signals continuation by checking the last word. The current SAGE backchannel system checks for "and", "but", "or", "because", and "so" for this purpose. Also, it can check the last word to see if it is appropriate as a reflective question, such as "And?" Presently, only the cue words listed are used for this reflective questioning, but there are others which could be added after some examination.

The system checks for some simple typographic parameters, such as if the user has started typing in their turn at all, or what sort of punctuation they have recently used. While such simple cues are not conceptually complex, they can be very useful for eliciting backchannel behavior. These checks allow the system to make certain prompts if the user has not started typing, or to include punctuation in determining if a person has finished typing.

A Sample Set of Backchannel and Turn-Taking Rules

A prototype set of backchannel and turn-taking rules (currently implemented and functioning) is included in Appendix A as an example of the backchannel program. The Lisp-style pseudocode in the Appendix bears some explanation, however.

The first rules deal with creating appropriate end of turn scenarios. The basic end of turn occurs when SAGE is satisfied that the input is sufficient:

```
(done (end-user-turn)
    ((not empty-input)
     (time 2 keystroke)
     (time 2 feedback)
     (not punctuation #\,)
     (not last-word "and" "but" "or" "because" "so")
     (input-satisfied)))
```

This rule ends the user's turn, but only triggers if the input criterion has been satisfied. A number of other conditions are checked for to help ensure that ending the turn is appropriate, namely that the user has entered something and that the last input was not a comma or a cue word. A two-second pause is also required, and a feedback pause is included to keep the SAGE from ending the turn right after prompting.

A second turn-ending rule ends the turn even if the user has not entered a satisfactory input, but requires a much longer and more obvious pause of four seconds. The SAGE input checker sometimes does not have a criterion to check the input against, and always returns failure, necessitating this rule. Once again, it checks a number of other conditions:

```
(done-unsatisfied (end-user-turn)
            ((not empty-input)
             (time 4 keystroke)
             (time 2 feedback)
             (not punctuation #\,)
             (not last-word "and" "but" "or" "because" "so")))
```

The third turn-ending rule checks to see if the typist has ended with a period. The combination of a period and a pause is considered sufficient to end the turn.

```
(period-end (end-user-turn)
        ((not empty-input)
         (time 2 keystroke)
         (time 2 feedback)
         (punctuation #\.)))
```

A fourth criteria that could be used to end the turn is the time since the user began. If the user takes too long, then the SAGE would interrupt after a shorter pause. This is not included in the Appendix, but would be a simple matter of adding this requirement to the sort of rule listed above (for a 14-second turn):

```
(time 14 user-turn)
```

The simplest backchannel is prompting during pauses, using various words as feedback. This can be done by directly checking the pauses in the rule.

```
(pause-feedback (word-feedback " Go on. " " Okay. " " Oy vey, continue. "
                        " Uh huh. ")
                ((not empty-input)
                 (time 2 keystroke)
                 (time 3 feedback)))
```

Another backchannel situation arises when the user doesn't start typing at the beginning of the turn, and should be prompted.

```
(prompt (word-feedback " Nu? " " Yes? " " Do not be uncomfortable. ")
        ((empty-input)
         (time 3keystroke)
         (time 3 feedback)))
```

Notice that the word-choice is obviously specific to the SAGE Rabbi character. While pause lengths are not obviously specific here, they would definitely alter if this set of rules was applied to a different personae. (A good example would be pauses between feedback in Japan, which are generally shorter than pause lengths within the U.S.)

A more complex situation is a response to a user's cue words. Typically when someone uses one of these cue words and pauses, it is an appropriate time to interject a comment. Here the comments are the cue words rephrased as questions, to prompt the person to type more:

```
(cue-word-feedback (ask-last-word)
                   ((time 1 keystroke)
                    (time 2 feedback)
                    (last-word "and" "but" "or" "because" "so")))
```

This short list of rules creates a set of complex behaviors, through interacting with the user and interacting with each other. While typing, it can be difficult to assign which response came from which rule or to predict when rules are triggered. An uncomfortable rule can be altered or removed by editing this set, and then tested again, creating a close feedback loop for the developer of the backchannel and turn-taking system.

Evaluation

First, the system as described above works. It does not interfere with the user's ability to type, because it is running as a background process and not taking very many computational cycles. It provides interactive feedback through the normal SAGE output channels while the user is typing, and ends the user's turn when the rule set dictates.

Some of the functionality of this system was not useful. The ability to end user control is not appropriate to SAGE because the user rarely if ever gains a control position in the SAGE interaction. Also, while it is possible to check a word against the entire input, this functionality is currently unused. The check could perhaps be used in the future to look for certain types of words, such as deictics or time words, but I have not looked into the possibility of using such information to determine appropriate feedback.

Even though I was expecting the interface to feel different with the feedback and turn-taking behavior, I was surprised by how different it felt. The turn-taking behavior especially adds a feeling of immediacy to the interaction, because (as in a normal conversation) the user will lose control if they stop producing communication. Thus, one has to pay close attention when interacting with the persona, something that is not true of most computer interactions. Also, the habitual response to short queries such as "ok" or "uh huh" is to move one's attention to the speaker of the short word or words, adding to the effect of the turn-taking.

In addition to demanding attention, the feedback creates a feeling of liveliness that is missing from typical typing interfaces. Getting reminders of the computer's attention every couple of seconds or during pauses really creates a feeling of a presence behind the screen, like occasional comments or simultaneous typing are used to establish a presence over the UNIX talk program.

Even though I wrote the rule set myself (and indeed the entire implementation), I still found myself unable to predict its behavior from situation to situation, or determine which rule caused a certain action. The reactions are too quick or depend on complex interactions between the timing parameters or cue parameters of various rules. I take this as a good sign. The level of interaction with the computer is moving out of the realm of the conscious and purposeful action and into the realm of habitual response, where discourse cues and responses actually occur. However, it meant I had to add extra diagnostics to be able to edit the rule set effectively.

A further level of complexity in the interaction surprised me. I found myself developing strategies to hold the turn. I knew that cue words and commas could be used to hold the turn, so I would put them in when I wanted to pause. My typing would occasionally look like, "I have a lot of work to do and it will take a lot of time, but when I finish it I will

be really happy." My development of turn-taking strategies is good in that in indicates a much more interactive input format, but the necessity of creating such strategies tells us that the system is still a far ways from being able to interpret the usual human-human turn-taking cues.

However, the monotony of the feedback tended to get annoying if the user decided to stop typing for some reason. Unlike people, the computer cannot interpret gaze to determine if the user has stopped paying attention. Also, the program is incapable of using semantic and intonational cues to determine the mode of the conversation, and thus cannot tell if the user has stopped listening.

Appropriate timing is difficult to determine. I would pick a time and test it. Often the same pause time would feel too long in one instance or too short in another. This is further confused because typing speed tends to run significantly slower than speaking speed, so pauses and pause relations that make sense in speaking do not map directly to typing. Determining when to give feedback, and integrating more cues into the decision, should be one of the next step in improving this backchannel system. Also, the timing should start adapting to the individual users, based on their typing speed or average typing pause length.

Future Research

A possible follow-up project to SAGE would connect certain backchannel and turn-taking behaviors to certain parts of the conversation, so that as the interaction changes modes from greeting to questioning to storytelling, the feedback system reflects each mode of conversation. Also, if the relevant parameters of the backchannel system can be determined (pause range and word choice among others), it may be possible to allow the children to modify backchannel and turn-taking behavior in the SAGE storytellers they

build. Such an introduction between discourse theory and children should be well-designed, but is well within the goals of SAGE project (Umaschi, 1996).

Pause timing could be improved in a number of ways. First, the system could have a number of interaction modes and the capability to switch between them. So, if the user does not type for five seconds, the computer could switch into an "ignored by user" mode, and slow down the rate of feedback. Second, an adaptive system such as the one designed by Klein could be used to have the interface adapt to a user's typing speed and pause lengths as they are working with the program. (Klein, 1996)

The backchannel system can currently search for words, but is unable to check for entire phrases. Detecting phrases that the user enters would allow the system to reply to certain phrases with appropriate quick feedback. This would be similar to the way it currently checks for and responds to cue words.

If time limitations are dealt with, then the system could get access to some more linguistic or semantic information. Linguistic or semantic cues (among other discourse cues) help shape backchannel and turn-taking, so a good system should be able to catch these cues. A part of speech tagger or dictionary lookup could be used to determine parts of speech. Similarly, a partial parsing process could be applied to input as it arrives to determine various grammatical relations between words. For instance, a lot of information can be gained simply by finding the tense, subject, or object of a verb. Perhaps this information could be used to find appropriate feedback or to help negotiate turn-taking. (One example is the use of part of speech information to determine whether it is appropriate to reflect the last word as a question. Verbs are more appropriate than nouns or adjectives. This would keep the program from replying "Now?" to the input "I am going to the store now," along with other simple mistakes. Another example would be checking to make sure the user is not in the middle of a verb phrase or other unfinished construction before end-

ing the user's turn.) A step down from this computationally intense work could be checks for wording around time, or deictic wording. In general, some mechanism should be written to make it easy to link higher-level discourse analysis into the system, and to deal with the analysis if it is too slow for the program to run while it is accepting input.

Now that a backchannel and turn-taking rule set has been written for a typing interface, it needs to be tested and modified, to determine things like exact pause timing. Also, testing should be done to determine how these relevant parameters change across situations and cultural backgrounds. Also, there will likely be a difference between novice and experienced computer users in their comfort level with the new interface, and this should be tested for. It may take the experienced users longer to adapt to the new interface, for instance.

Also, there is a lack of research on conversational backchannel and turn-taking between people, especially in cross-cultural comparison. This research would allow a much more informed exploration of interactions affected by people's expectations of interpersonal discourse, such as human-computer interaction.

Conclusion

While the basic computer interface paradigm of dyadic input/output messaging is considered by many to be a settled and proven matter, this textual messaging system largely fails to take into account user expectations of the computer, especially in cases where a program assumes a persona or character of some sort. In particular, there is a problem with the lack of responsiveness while a user is typing in text, and this problem can be partially fixed by some relatively minor programming around the text input paradigm.

As demonstrated by the relative simplicity of the coding needed for this project, adding a basic set of backchannel and turn-taking rules into an interface is a feasible task both

in terms of coding time and processing time. This basic set of discourse rules improves the textual input interface in a number of ways. First, it reduces the traditional dependence on the return key as "sending" input to the program, breaking down the back-and-forth messaging paradigm to a large extent. Second, both the backchannel and turn-taking behavior add a certain responsiveness or feeling of life to the computer interface, making it feel drastically different from the traditionally unresponsive or "dead" input activity. This new "feel" of a traditional input form (typing) makes the SAGE program feel more "alive", by animating the persona of a SAGE avatar. Third, backchannel behavior can be used to encourage, discourage, or direct user input while the user is entering it, much in the same way that people will keep a continuous line of communication open during conversation, no matter who is actually doing the speaking at any moment.

The domain of backchannel and turn-taking behavior is of course highly applicable to multimodal agents such as Gandalf, and any other system seeking to obtain speech input from a person. But this domain is also relevant to many other forms of input, even though they do not include the speech recognition that is generally considered necessary for a simulation of conversational behavior. This relevance is partially due to the tendency of people to anthropomorphize the computer, and to go out of their way looking for human-like characteristics in communication with a computer. Even in the supposedly well-established area of typing input, simple application of this domain sharply alters the feel of an interface.

Cultural, situational, and personal differences in backchannel and turn-taking expectations form one of the major hurdles in designing these systems, as each instance of a backchannel or turn-taking program could demand a slightly different set of behaviors. This was well-demonstrated by the situation with the SAGE characters (which are culturally diverse) from the beginning of this project. One easy solution to this dilemma, as demon-

strated by the approach of this project, is to create a very flexible base system which is easily modifiable from one persona or situation to another, and perhaps even within the same interaction.

Timing was more difficult than expected, due to personal and situational differences in typing speed and pause style, and the lack of many cues used for timing in normal conversation. More development is required before feedback and turn-ending occur at the proper juncture. A couple of ideas are listed below.

While it is easy to put together a system which can do surface backchannel, actually giving feedback that recognizes and uses the content of what is being entered is a very difficult problem, and likely will not be solved for a long time. However, until this area is investigated much more thoroughly, computers will not be able to effectively reproduce person-to-person backchannel behavior. Similarly, moving the computer away from the discourse situation addressed in this paper will create entirely new problems. For example, if the computer is one participant in a many-person conversation, how does it start to address the (much more complex) turn-taking dynamics of the situation?

Acknowledgments

My eternal gratitude to Marina Umaschi, who designed the SAGE system that I built on, and guided me through this thesis. Also, a heartfelt thank you to Professor Justine Cassell, who taught me all of the discourse theory I know and put up with my foibles for the year that I've been in her research group. All the other people in the Gesture and Narrative Language group at the MIT Media Lab have been very supportive, and without them this would have been much more difficult: Jennifer Glos, Hannes Viljalmsson, Obed Torres, Nick Montfort, Scott Prevost, Matthew Sakai, and Erin Panttaja.

References

Cassell, J., C.Pelachaud, N.I. Badler, M. Steedman, B. Achorn, T. Becket, B. Douville, S. Prevost, M. Stone. (1994) "ANIMATED CONVERSATION: Rule-Based Generation of Facial Expression, Gesture, and Spoken Intonation for Multiple Conversation Agents." Siggraph '94, Orlando, USA.

Cherny, L., (1997) "The MUD Register: Conversational Modes of Action in a Text-Based Virtual Reality." Submitted to The Program in Media Arts and Sciences, Dec 1997.

Foner, L. N. (1993) "Agents Memo 93-01", Agents Group, MIT Media Lab.

Goffman, E. (1983) *Forms of Talk.* Philadelphia, PA: University of Philadephia Press.

Goodwin, C. (1981) *Conversational Organization: Interaction Between Speakers and Hearers.* New York, NY: Academic Press.

Klein, J. (1996) "'Speak as You are Spoken To': A software agent that learns aspects of turn-taking and other elements of conversational style." Final Project Paper, *Discourse* class, MIT Media Laboratory.

Markowitz, (1996) J. A. *Using Speech Recognition.* Upper Saddle River, NJ: Prentice-Hall, Inc.

Maudlin, M. L. (1994) "Chatterbots, Tinymuds, and the Turing Test: Entering the Loebner Prize Competition." 12th Conference on AI.

Moon, Y., C. I. Nass. (1997) "How 'Real' Are Computer Personalities? Psychological responses to personality types in human-computer interaction." Communication Research (in press)

Nass, C. I, Y. Moon, N. Green. (1997) "How Powerful Are Gender Stereotypes? An experimental demonstration." Journal of Applied Social Psychology (in press).

Sacks, H., E. A. Schegloff, G.A. Jefferson. (1974) "A Simplest Schematics for the Organization of Turn-Taking in Conversation." Language, 50, pp. 696-735.

Schegloff, E. A. (1982) "Discourse as an Interactional Acheivement: Some Uses of 'uh huh' and Other Things that Come Between Sentences." In Deborah Tannen, (eds.) Georgetown University Round Table on Languages and Linguistics 1981:71-93. Washington, D.C.: Georgetwon University Press.

Schiffrin, D. (1987) *Discourse Markers*. Cambridge, MA:Cambridge University Press.

Thorisson, K. R., J. Cassell. (1996) "Why Put an Agent in a Body: The Importance of Communicative Feedback in Human-Humanoid Dialogue." Presented at Lifelike Computer Characters.

Thorisson, K. R. (1996) "Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills." Submitted to The Program in Media Arts and Sciences.

Umaschi, M. (1997) "Soft Interfaces for Interactive Storytelling: Learning about Identity and Communication." Submitted to The Program in Media Arts and Sciences.

Umaschi, M. (1996) "Soft Interfaces for Personal Stories: Storytelling Systems for Children's Learning of Self and Communication." Proposal submitted to The Program in Media Arts and Sciences.

Weizenbaum, J. (1966) *ELIZA - a computer program for the study of natural language communication between man and machine*. Communications of the ACM 9.

Whittaker, S., P. Stenton. (1988) "Cues and Control in Expert-Client Dialogues." Proceedings of the 26th Annual Meeting of the Association for Computational Linguistics.

Appendix A: Backchannel and Turn-Taking Sample Rule Set

```
;; first, appropriate end-of-turn
;; don't end on a continuing cue word... or a comma
(done (end-user-turn)
    ((not empty-input)
     (time 2 keystroke)
     (time 2 feedback)
     (not punctuation #\,)
     (not last-word "and" "but" "or" "because" "so")
     (input-satisfied)))

;; end-of-turn when input conditions haven't been (necessarily) satisfied
(done-unsatisfied (end-user-turn)
            ((not empty-input)
             (time 4 keystroke)
             (time 2 feedback)
             (not punctuation #\,)
             (not last-word "and" "but" "or" "because" "so")))

;; if they don't say anything, prompt them
(prompt (word-feedback " Nu? " " Yes? " " Do not be uncomfortable. ")
      ((empty-input)
       (time 4 keystroke)
       (time 3 feedback)))

;; if they have ended with a cue word, ask it back to them
(cue-word-feedback (ask-last-word)
            ((time 1 keystroke)
             (time 2 feedback)
             (last-word "and" "but" "or" "because" "so")))

;; general pause feedback - if they stop, prompt them
(pause-feedback (word-feedback " Go on. " " Okay. " " Oy vey, continue. "
                  " Uh huh. ")
          ((not empty-input)
           (time 2 keystroke)
           (time 3 feedback)))

;;; sometimes people indicate end with period - check here
(period-end (end-user-turn)
```

```
((not empty-input)
 (time 2 keystroke)
 (time 2 feedback)
 (punctuation #\.)))
```

Appendix B: Lisp Code for the Backchannel and Turn-Taking System

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;backchannel.lisp
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass backchannel ()
  ((interpreter :accessor interpreter :initarg :interpreter :initform nil)
   (backchannel-event-list :accessor bevent-list :initform nil)
   (backchannel-process :accessor bprocess :initform nil)
   (back-process-kill :accessor bprocess-kill :initform nil)
   (state :accessor state :initform (make-instance 'backchannel-state))
   (callbacks :accessor callbacks :initform
    (make-instance 'backchannel-callbacks))))

(defclass backchannel-callbacks ()
  ((end-user-turn :accessor end-user-turn :initform nil)
   (end-user-control :accessor end-user-control :initform nil)
   (send-output :accessor send-output :initform nil)))

(defclass backchannel-state ()
  ((original-time :accessor original-time :initform 0)
   (last-time-checked :accessor last-time-checked :initform 0)
   (current-time :accessor current-time :initform 0)
   (feedback-time :accessor feedback-time :initform 0)
   (typing-start-time :accessor typing-start-time :initform 0)
   (control-start-time :accessor control-start-time :initform 0)
   (keystroke-handoff :accessor keystroke-handoff :initform nil)
   (keystroke-time :accessor keystroke-time :initform 0)
   (user-typing :accessor user-typing :initform nil)
   (user-control :accessor user-control :initform nil)
    ;; does the user have control?
   (current-input-req :accessor current-input-req :initform nil)
   (current-input :accessor current-input :initform nil)
   (current-input-parsed :accessor current-input-parsed :initform nil)
   (punctuation :accessor punctuation :initform
      '(#\, #\. #\' #\" #\( #\) #\? #\!))
   (whitespace :accessor whitespace :initform
      '(#\Space #\Tab #\Newline))))
```

```
(defclass backchannel-action ()
  ((name :accessor name :initarg :name :initform "")
   (activated :accessor activated :initform t)
   (backchannel-req-list :accessor breqs :initarg :breqs :initform nil)))

(defclass back-action-words (backchannel-action)
  ((word-list :accessor word-list :initarg :word-list :initform nil)))
    ;; word-list is currently just a list of strings

(defclass back-action-turn (backchannel-action) ())
    ;; ends the user turn

(defclass back-action-control (backchannel-action) ())
    ;; ends the user control

(defclass ask-last-word (backchannel-action) ())
    ;; rephrases the last word as a question: "because" --> "Because?"


(defclass backchannel-req ()
  ((negate :accessor negate :initarg :not :initform nil)))

(defclass time-req (backchannel-req)
  ((tim :accessor tim :initarg :time :initform 0)
     ;; the time involved
   (keystroke :accessor keystroke :initform nil)
   (feedback :accessor feedback :initform nil)
   (user-turn :accessor user-turn :initform nil)
   (user-control :accessor user-control :initform nil)))

(defclass punctuation-req (backchannel-req)
  ((character-list :accessor character-list :initarg :character-list
                   :initform nil)))

(defclass correct-input-req (backchannel-req) ())

(defclass word-req (backchannel-req)
  ((word-list :accessor word-list :initarg :word-list :initform nil)))

(defclass last-word-req (backchannel-req)
  ((word-list :accessor word-list :initarg :word-list :initform nil)))
```

```
(defclass empty-input-req (backchannel-req) ())



;;;;;;;;;;;;;;;;;;;;;;;;;;;; Package Interface ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; creation and action manipulation

(defun stream-to-backchannel (stream &optional interpreter)
  (let ((the-back (make-instance 'backchannel :interpreter interpreter)))
    (when (not (null stream))
      (setf (bevent-list the-back) (stream-to-backchannel-list stream)))
    the-back))

(defun create-backchannel-system (interpreter &key end-user-turn
                                                    end-user-control
                                                    send-output
                                                    stream backchannel)
  (let ((the-back (make-instance 'backchannel :interpreter interpreter)))
    (when (not (null stream))
      (setf (bevent-list the-back) (stream-to-backchannel-list stream)))
    (when (not (null backchannel))
      (setf (interpreter backchannel) interpreter)
      (setf the-back backchannel))
    (when (not (null end-user-turn))
      (setf (end-user-turn (callbacks the-back)) end-user-turn))
    (when (not (null end-user-control))
      (setf (end-user-control (callbacks the-back)) end-user-control))
    (when (not (null send-output))
      (setf (send-output (callbacks the-back)) send-output))
    the-back))

(defmethod get-action-list ((b backchannel))
  (copy-list (bevent-list b)))

(defmethod get-action-name-list ((b backchannel))
  (mapcar #'(lambda (elem)
              (name elem))
          (bevent-list b)))

(defmethod add-action ((b backchannel) &key stream
     input action name action-list)
  (when (not (null stream))
```

```
  (setf (bevent-list b)
        (append (bevent-list b)
                (stream-to-backchannel-list stream))))
 (when (not (null input))
  (setf (bevent-list b)
        (append (bevent-list b) (list (input-to-backchannel input)))))
 (when (not (null action))
  (setf (bevent-list b)
        (append (bevent-list b) (list action))))
 (when (not (null action-list))
  (mapcar #'(lambda (act)
              (add-action b :action act))
          action-list))
 (when (not (null name))
  (activate-action b name)))

(defmethod delete-action ((b backchannel) &key action name
                          action-list)
 (when (not (null action))
  (setf (bevent-list b) (filter (bevent-list b)
                                #'(lambda (elem)
                                    (not (eq action elem))))))
 (when (not (null action-list))
  (mapcar #'(lambda (act)
              (delete-action b :action act))
          action-list))
 (when (not (null name))
  (setf (bevent-list b) (filter (bevent-list b)
                                #'(lambda (elem)
                                    (not (eq name (name elem))))))))

(defmethod deactivate-action ((b backchannel)
   &key action name action-list)
 (when (not (null action))
  (setf (activated action) nil))
 (when (not (null action-list))
  (mapcar #'(lambda (act)
              (deactivate-action b :action act))
          action-list))
 (when (not (null name))
  (mapcar #'(lambda (elem)
              (when (eq (name elem) name)
               (setf (activated elem) nil)))
```

```
        (bevent-list b))))

(defmethod activate-action ((b backchannel) &key action name action-list)
  (when (not (null action))
    (setf (activated action) t))
  (when (not (null action-list))
    (mapcar #'(lambda (act)
                (activate-action b :action act))
            action-list))
  (when (not (null name))
    (mapcar #'(lambda (elem)
                (when (eq (name elem) name)
                  (setf (activated elem) t)))
            (bevent-list b))))

;; interaction with running interpreter

; calls into backchannel system

(defmethod run-backchannel ((b backchannel))
  (setf (bprocess-kill b) nil)
  (setf (original-time (state b)) (get-local-time))
  (setf (user-typing (state b)) nil)
  (setf
   (bprocess b)
   (process-run-function
    "backchannel"
    #'run-backchannel-background b)))

(defmethod user-start-control ((b backchannel))
  (setf (user-control (state b)) t)
  (setf (control-start-time (state b)) (get-local-time))
  )

(defmethod user-start-typing ((b backchannel) current-input-fun)
  (setf (typing-start-time (state b)) (get-local-time))
  (setf (keystroke-time (state b)) (get-local-time))
  (setf (current-input (state b)) "")
  (setf (current-input-parsed (state b)) nil)
  (setf (current-input-req (state b)) current-input-fun)
  (setf (user-typing (state b)) t))

(defmethod keystroke-callback ((b backchannel) current-text)
```

```
  (setf (keystroke-time (state b)) (get-local-time))
  (setf (current-input (state b)) current-text)
  (setf (current-input-parsed (state b)) (parse-to-word-list b current-text)))

(defmethod user-stop-typing ((b backchannel))
  (setf (user-typing (state b)) nil))

(defmethod user-stop-control ((b backchannel))
  (setf (user-control (state b)) nil))

(defmethod stop-backchannel ((b backchannel))
  (setf (bprocess-kill b) t))

;;;;;;;;;;;;;;;;;;;;;;;;; Backchannel Watchdog Process ;;;;;;;;;;;;;;;;;;;;;;;;;

(defun run-backchannel-background (b)
  (loop
    (when (or (not (user-typing (state b)))
              (not (time-check b)))
      (process-wait "user input disabled" #'backchannel-background-wait b))
    (when (bprocess-kill b) (return-from run-backchannel-background nil))
    (run-requirements b)
    (setf (last-time-checked (state b)) (get-local-time))))

(defun backchannel-background-wait (b)
  (or (and (time-check b) (user-typing (state b))) (bprocess-kill b)))

(defun time-check (b)
  (> (get-local-time) (+ 1 (last-time-checked (state b)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;; Saving and Loading ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun stream-to-backchannel-list (stream)
  (let ((back-list '()))
    (do ((inp (read stream nil nil) (read stream nil nil)))
        ((null inp) (reverse back-list))
      (setf back-list (cons (input-to-backchannel inp) back-list)))))

(defun input-to-backchannel (inp)
  (dassert (= 3 (length inp)) "in input-to-backchannel: wrong length input")
  (let ((the-action (translate-action (second inp))))
    (setf (name the-action) (first inp))
    (setf (breqs the-action)
```

```
        (mapcar #'(lambda (imp)
                    (translate-req imp))
                (third inp)))
    the-action))


;;;;;;;;;;;;;;;;;;; Requirement Checking ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defmethod run-requirements ((b backchannel))
  (update-state b)
  (dolist (action (bevent-list b))
    (run-action b action))
  )

(defmethod update-state ((b backchannel))
  (setf (current-time (state b)) (get-local-time)))

(defmethod run-action ((b backchannel) action)
  (let ((reqs-satisfied t))
    (dolist (req (breqs action))
      (when (not (check-req-not req b))
        (setf reqs-satisfied nil)
        (return nil)))
    (when reqs-satisfied (do-action action b))))

(defmethod do-action ((a back-action-words) b)
  (when (not (null (send-output (callbacks b))))
    (setf (feedback-time (state b)) (get-local-time))
    (funcall (send-output (callbacks b)) (choose-random (word-list a)))))

(defmethod do-action ((a ask-last-word) b)
  (when (and (not (null (send-output (callbacks b))))
             (not (null (current-input-parsed (state b)))))
    (setf (feedback-time (state b)) (get-local-time))
    (funcall
     (send-output (callbacks b))
     (concatenate
      'string
      " "
      (string-upcase
       (first (last (current-input-parsed (state b))))
       :start 0 :end 1)
      "? "))))
```

```
(defmethod do-action ((a back-action-turn) b)
 (when (not (null (end-user-turn (callbacks b))))
  (setf (user-typing (state b)) nil)
  (funcall (end-user-turn (callbacks b)))))

(defmethod do-action ((a back-action-control) b)
 (when (not (null (end-user-control (callbacks b))))
  (setf (user-typing (state b)) nil)
  (setf (user-control (state b)) nil)
  (funcall (end-user-control (callbacks b)))))

(defmethod check-req-not ((r backchannel-req) b)
 (if (negate r)
  (not (check-req r b))
  (check-req r b)))

(defmethod check-req ((r time-req) b)
 (let* ((critical-time (- (current-time (state b)) (tim r))))
  (when (not (null (keystroke r)))
   (when (not (> critical-time (keystroke-time (state b))))
    (return-from check-req nil)))
  (when (not (null (feedback r)))
   (when (not (> critical-time (feedback-time (state b))))
    (return-from check-req nil)))
  (when (not (null (user-turn r)))
   (when (not (> critical-time (typing-start-time (state b))))
    (return-from check-req nil)))
  (when (not (null (user-control r)))
   (when (not (> critical-time (control-start-time (state b))))
    (return-from check-req nil)))
  t))

(defmethod check-req ((r punctuation-req) b)
 (let ((last-non-whitespace (get-last-char b (current-input (state b)))))
  (member last-non-whitespace (character-list r) :test #'eql)))

(defmethod check-req ((r correct-input-req) b)
 (if (null (current-input-req (state b)))
  t
  (funcall (current-input-req (state b)) (current-input (state b)))))

(defmethod check-req ((r word-req) b)
```

```
  (let ((the-input-words (current-input-parsed (state b))))
   (dolist (a-word (word-list r) nil)
    (when (member a-word the-input-words :test #'equalp)
     (return-from check-req t)))))

(defmethod check-req ((r last-word-req) b)
 (let ((the-input-words (current-input-parsed (state b))))
  (when (not (null the-input-words))
   (when (member
    (first (last the-input-words))
    (word-list r) :test #'equalp) t)))

(defmethod check-req ((r empty-input-req) b)
 (null (current-input-parsed (state b))))

;;;;;;;;;;;;;;;; Action and Requirement Translation ;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun translate-action (inp)
 (dassert (<= 1 (length inp)) "in translate-action: empty input")
 (when (eq 'word-feedback (first inp))
  (dassert (<= 2 (length inp)) "in translate-action: no words")
  (return-from translate-action
   (make-instance 'back-action-words :word-list (rest inp))))
 (when (eq 'end-user-turn (first inp))
  (return-from translate-action
   (make-instance 'back-action-turn)))
 (when (eq 'end-user-control (first inp))
  (return-from translate-action
   (make-instance 'back-action-control)))
 (when (eq 'ask-last-word (first inp))
  (return-from translate-action
   (make-instance 'ask-last-word)))

 (dassert nil "in translate-action: unknown action type"))

(defun translate-req (inp)
 (dassert (<= 1 (length inp)) "in translate-req: empty input")
 (let ((new-inp (if (eq 'not (first inp)) (rest inp) inp))
       (negation (eq 'not (first inp))))
  (dassert (<= 1 (length new-inp)) "in translate-req: empty input")
  (when (eq 'time (first new-inp))
   (dassert (<= 3 (length new-inp)) "in translate-req: short time input")
   (dassert (numberp (second new-inp)) "in translate-req: bad time input")
```

```
 (let ((the-req (make-instance 'time-req
                  :time (second new-inp) :not negation))
       (type-list (nthcdr 2 new-inp)))
   (when (not (null (member 'keystroke type-list)))
     (setf (keystroke the-req) t))
   (when (not (null (member 'feedback type-list)))
     (setf (feedback the-req) t))
   (when (not (null (member 'user-turn type-list)))
     (setf (user-turn the-req) t))
   (when (not (null (member 'user-control type-list)))
     (setf (user-control the-req) t))
   (return-from translate-req the-req)))
(when (eq 'punctuation (first new-inp))
 (dassert
  (<= 2 (length new-inp))
  "in translate-req: short punctuation input")
 (mapcar #'(lambda (char)
             (dassert (characterp char)
                      "in translate-req: bad character for punctuation"))
         (rest new-inp))
 (return-from translate-req
   (make-instance 'punctuation-req :character-list (rest new-inp)
                  :not negation)))
(when (eq 'input-satisfied (first new-inp))
 (return-from translate-req
   (make-instance 'correct-input-req :not negation)))
(when (eq 'word (first new-inp))
 (dassert (<= 2 (length new-inp)) "in translate-req: short word input")
 (mapcar #'(lambda (word)
             (dassert (stringp word)
                      "in translate-req: bad character for punctuation"))
         (rest new-inp))
 (return-from translate-req
   (make-instance 'word-req :word-list (rest new-inp)
                  :not negation)))
(when (eq 'last-word (first new-inp))
 (dassert (<= 2 (length new-inp)) "in translate-req: short word input")
 (mapcar #'(lambda (word)
             (dassert (stringp word)
                      "in translate-req: bad character for punctuation"))
         (rest new-inp))
 (return-from translate-req
   (make-instance 'last-word-req :word-list (rest new-inp)
```

```
                :not negation)))
  (when (eq 'empty-input (first new-inp))
   (return-from translate-req
    (make-instance 'empty-input-req :not negation)))


  (dassert nil "in translate-req: unknown requirement")))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; utilities ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun choose-random (lst)
 (nth (random (length lst)) lst))

(defun get-local-time ()
 (/ (get-internal-real-time) internal-time-units-per-second))

(defmethod get-last-char ((b backchannel) strng)
 (let ((last-char (- (length strng) 1)))
  (loop
   (when (> 0 last-char) (return-from get-last-char nil))
   (when (not (member (char strng last-char)
                      (whitespace (state b))))
    (return-from get-last-char (char strng last-char)))
   (setf last-char (- last-char 1)))))

(defmethod parse-to-word-list ((b backchannel) strng)
 (let ((len (length strng))
       (begin-index 0)
       (end-index 0))
  (loop
   (when (<= len begin-index) (return-from parse-to-word-list nil))
   (when (not (or (member (char strng begin-index)
                          (whitespace (state b)))
                  (member (char strng begin-index)
                          (punctuation (state b)))))
    (return nil))
   (setf begin-index (+ begin-index 1)))
  (setf end-index (+ 1 begin-index))
  (loop
   (when (<= len end-index)
    (return-from parse-to-word-list
     (list (subseq strng begin-index))))
```

```
(when (or (member (char strng end-index)
                  (whitespace (state b)))
          (member (char strng end-index)
                  (punctuation (state b))))
  (return-from parse-to-word-list
    (cons (subseq strng begin-index end-index)
          (parse-to-word-list b (subseq strng end-index)))))
(setf end-index (+ end-index 1)))))
```