Chapter 20

Feasibility Studies for Programming in Natural Language

HENRY LIEBERMAN and HUGO LIU
Media Laboratory, Massachusetts Institute of Technology, {lieber, hugo}@media.mit.edu

Abstract. We think it is time to take another look at an old dream—that one could program a computer by speaking to it in natural language. Programming in natural language might seem impossible, because it would appear to require complete natural language understanding and dealing with the vagueness of human descriptions of programs. But we think that several developments might now make programming in natural language feasible. First, improved broad coverage natural language parsers and semantic extraction techniques permit partial understanding. Second, mixed-initiative dialogues can be used for meaning disambiguation. And finally, where direct understanding techniques fail, we hope to fall back on Programming by Example, and other techniques for specifying the program in a more fail-soft manner. To assess the feasibility of this project, as a first step, we are studying how non-programming users describe programs in unconstrained natural language. We are exploring how to design dialogs that help the user make precise their intentions for the program, while constraining them as little as possible.

Key words. natural language programming, natural language processing, parsing, part-of-speech tagging, computer science education, programming languages, scripting languages, computer games.

1. Introduction

We want to make computers easier to use and enable people who are not professional computer scientists to be able to teach new behavior to their computers. The Holy Grail of easy-to-use interfaces for programming would be a natural language interface—just *tell* the computer what you want! Computer science has assumed this is impossible because it would be presumed to be "AI Complete"—require full natural language understanding.

But our goal is not to enable the user to use completely unconstrained natural language for any possible programming task. Instead, what we might hope to achieve is to attain enough partial understanding to enable using natural language as a communication medium for the user and the computer to cooperatively arrive at a program, obviating the need for the user to learn a formal computer programming language. Initially, we will work with typed textual input, but ultimately we would hope for a spoken language interface, once speech recognizers are up to the task. We are now evaluating commercially available speech recognizers, and are developing new techniques for correction of speech recognition errors based on Common Sense knowledge (Stocky et al., 2004).

We believe that several developments might now make natural language programming possible where it was not feasible in the past.

Henry Lieberman et al. (eds.), End User Development, 459–473.

- *Improved language technology.* While complete natural language understanding still remains out of reach, we think that there is a chance that recent improvements in robust broad-coverage parsing (Collins, 2003) (MontyLingua), semantically informed syntactic parsing and chunking, and the successful deployment of natural language command-and-control systems (Liu, 2003) might enable enough partial understanding to get a practical system off the ground.
- *Mixed-initiative dialogue.* We do not expect that a user would simply "read the code aloud." Instead, we believe that the user and the system should *have a conversation* about the program. The system should try as hard as it can to interpret what the user chooses to say about the program, and then ask the user about what it doesn't understand, to supply missing information, and to correct misconceptions.
- *Programming by Example*. We will adopt a *show and tell* methodology, which combines natural language descriptions with concrete example-based demonstrations. Sometimes it is easier to demonstrate what you want then to describe it in words. The user can tell the system "here is what I want," and the system can verify its understanding with "Is this what you mean?" This will make the system more *fail-soft* in the case where the language cannot be directly understood, and, in the case of extreme breakdown of the more sophisticated techniques, we will simply allow the user to type in code.

2. Feasibility Study

We were inspired by the Natural Programming Project of Pane and Myers at Carnegie-Mellon University (Pane et al., 2001). Pane and Myers (2004) conducted studies asking non-programming fifth-grade users to write descriptions of a Pac-Mac game (in another study, college students were given a spreadsheet programming task). The participants also drew sketches of the game so they could make deictic references.

Pane and Myers then analyzed the descriptions to discover what underlying abstract programming models were implied by the users' natural language descriptions. They then used this analysis in the design of the HANDS programming language. HANDS uses a direct-manipulation, demonstrational interface. While still a formal programming language, it hopefully embodies a programming model that is closer to users' "natural" understanding of the programming process before they are "corrupted" by being taught a conventional programming language. They learned several important principles, such as that users rarely referred to loops explicitly, and preferred event-driven paradigms.

Our aim is more ambitious. We wish to directly support the computer understanding of these natural language descriptions, so that one could do "programming by talking" in the way that these users were perhaps naively expecting when they wrote the descriptions.

As part of the feasibility study, we are transcribing many of the natural language descriptions and seeing how well they will be handled by our parsing technology (Liu, online). Can we figure out where the nouns and verbs are? When is the user talking about a variable, loop, or conditional?

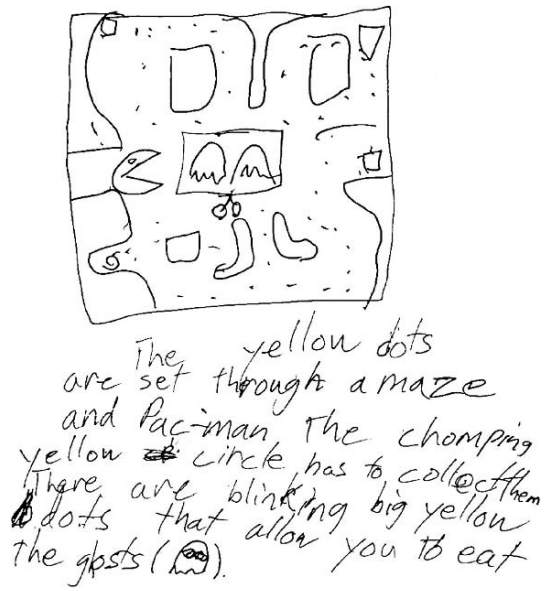

Figure 20.1. Non-programmer's description of Pac-Man.

One of our guiding principles will be to abandon the programming language dogma of having a single representation for each programming construct. Instead, we will try to collect as many verbal representations of each programming construct as we can, and see if we can permit the system to accept *all* of them.

3. Designing Dialogue Management for Natural Language Programming

The paradigm for the user interaction should reproduce the situation of the user talking to a more knowledgeable programmer, who knows all the implementation techniques. The goal for the user is to communicate their intent.

We have been inspired by some of the dialogue management philosophy of Collagen (Rich et al., 2001). Collagen uses a discourse model to support a user and an on-screen agent cooperatively working on a task. Either the user or the system can take initiative at any point, and the system keeps track of such dialogue management issues as which goals are being worked on and which are pending.

We envision a possible dialogue as follows, inspired by the Pac-Man example above,

User: We are going to program a game, Pac-Man.
System: OK. I know a few things about how to make games. We need to decide what the playing field looks like, design some characters for the game, how the users are going to control the characters, and how the scoring system is going to work. OK?

U: Sure. The yellow dots are set through a maze, and Pac-Man, the chomping yellow circle, has to collect them (see the user's sketch in Figure 20.1).

S: OK, let me see if I have got this straight. You want the playing field to be a grid of squares, each containing a yellow dot, right? (System displays a grid of squares with yellow dots). To make it a maze, we need to have some walls. Can you draw some walls for me?

(1). U: (User draws in walls, selecting color, texture, etc.)

S: Pac-Man is the name of a character in the game, represented as a yellow circle, is that right? How is this? (System displays a yellow circle in one of the grid squares).

(2). U: Oh, the dots should be red. (Things do not have to be done "in order." You can refer to something previously said.)

S: OK. What does it mean for Pac-Man to "collect" a dot?

U: Whenever Pac-Man is on a square with a dot, he eats it. That means that the dot is gone and the player gets another point. (The system introduces a variable to count the number of dots eaten by Pac-Man).

S: OK, now I do not understand what "eats" means. Can you explain it?

U: When you draw Pac-Man, you alternate between a picture of Pac-Man with his mouth open and a picture of Pac-Man with his mouth closed.

S: Can you draw me what Pac-Man looks like when he has his mouth open? Mouth closed?

(3). U: (Draws.) There are blinking big yellow dots that allow you to eat the ghosts (draws picture of ghost).

S: (Continues dialogue about dots and ghosts . . . .)

Below, we give some analysis of the above-presented dialog. From (1), the system must recognize that a "game" is the top-level programming goal. To conduct the rest of the dialog, we assume the system has some script- or frame-based knowledge about the major programming subgoals in programming a game. It should leverage this understanding to dually know what to expect from the user, and to act as a tutor to guide the user.

As (3) demonstrates, users will attempt to convey a lot of information all at once. It is the job of the language understanding system to identify major intended *actions* (e.g., "set through"), each of which are associated with a thematic *agent* role (e.g., "the yellow dots"), and a thematic *patient* role (e.g., "a maze"). The system will also try to correlate these filled role slots with its repertoire of programming tricks. For example, in (3), "yellow dots" might be visual primitives, and "a maze" might invoke a script about how to construct such a structure on the screen and in code. In (4), the dialog management system reconfirms its interpretation to the user, giving the user the opportunity to catch any glitches in understanding.

In (5), the system demonstrates how it might mix natural language input with input from other modalities as required. Certainly we have not reached the point where good graphic design can be dictated in natural language! Having completed the maze layout

subgoal, the system planning agency steps through some other undigested information gleaned from (3). In (6), it makes some inference that Pac-Man is a character in this game based on its script knowledge of a game.

Again in (9), the user presents the system with a lot of new information to process. The system places the to-be-digested information on a stack and patiently steps through to understand each piece. In (10), the system does not know what "eats" should do, so it asks the user to explain that in further detail. And so on.

While we may not be able to ultimately achieve all of the language understanding implied in the example dialogue above, and we may have to further constrain the dialogue, the above example does illustrate some important strategies, including iterative deepening of the program's understanding (Lieberman and Liu, 2002).

4. Designing Natural Language Understanding for Programming

Constructing a natural language understanding system for programming must be distinguished from the far more difficult task of open domain story understanding. Luckily, natural language understanding for programming is easier than open domain story understanding because the discourse in the programming domain is variously constrained by the task model and the domain model. This section attempts to flesh out the benefits and challenges which are unique to a language understanding system for programming.

4.1. CONSTRAINTS FROM AN UNDERLYING SEMANTIC MODEL

The language of discourse in natural language programming is first and foremost, constrained by the underlying semantic model of the program being constructed. Consider, for instance, the following passage from a fifth-grader non-programmer's description of the Pacman game:

> *Pacman eats a big blink dot and then the ghosts turn blue or red and pacman is able to eat them. Also his score advances by 50 points.*

In the previous section, we argued that through mixed-initiative dialogue, we can begin to progressively disambiguate a programmatic description like the one shown above, into an underlying semantic model of a *game*. Establishing that "Pacman" is the main character in the game helps us to parse the description. We can, for example, recognize that the utterances "Pac man" and "pacman" probably both *refer* to the character "Pacman" in the game, because both are lexicographically similar to "Pacman," but there is also the confirming evidence that both "Pac man" and "pacman" take the action "to eat," which is an action typically taken by an agent or character. Having *resolved* the meanings of "Pac man" and "pacman" into the character "Pacman," we can now resolve "his" to "Pacman" because "his" refers to a single agent, and "Pacman" is the only plausible referent in the description. We can now infer that "eat" refers to an ability of the agent "Pacman," and "score" is a member variable associated with "Pacman," and that the score has the ability to advance, and so on and so forth.

In summary, the underlying semantic model of a program provides us with *unambiguous referents* that a language understanding system can parse text into. All levels of a language processing system, including speech recognition, semantic grouping, part-of-speech tagging, syntactic parsing, and semantic interpretation, benefit from this phenomena of *reference*. Although the natural language input is ideally unconstrained, the semantic model we are mapping into is well-constrained. Language resolution also has a nice cascading effect, which is, the more resolutions you make, the more you are able to make (by leveraging existing "*islands of certainty*"). Resolving "Pac man" and "pacman" in turn allows us to resolve "his" and these in turn allow us to resolve "eat" and "score." Of course, in our proposed mixed-initiative model, we can always prompt the user for confirmation of any ambiguities which cannot be resolved.

In the above example, we discuss how objects and actions get resolved, but what about programmatic controls? Are these easy to recognize and resolve? By studying the "programming by talking" styles of many users, we expect to be able to identify a manageable set of salient keywords, phrases, and structures which indicate programmatic controls like conditionals and loops. Although, it would come as no surprise if "programming by talking" maps somewhat indirectly rather than directly onto programming control structures. For example, in the usability studies of Pane and Myers, it is uncommon to find explicit language to describe loops directly. Instead, there is evidence for natural language descriptions mapping into *implicit loop operations* in the form of Lisp-style list processing functions like "map," "filter," and "reduce." For example, the utterance, "*Pacman tries to eat all the big blinking dots*" does not seem like a programmatic control, but it actually expresses several loops implicitly (and quite elegantly, as we might add). We can paraphrase the utterance in pseudo-code as follows:

```
map(Pacman.eat,
      filter(lambda dot:
            dot.big AND dot.blinking,
            dots))
```

We are aided in the generation of this pseudo-code interpretation by knowledge of the preferences/constraints of the underlying semantic model, i.e., something that Pacman can do is eat (*x.y ()* relation), a dot is something which can be eaten (*x(y)* relation), and dots can be big and blinking ($x.y$ relations).

Thus far, we have generally outlined a strategy for mapping a programmatic description into a code model by progressive stages of semantic resolution, but we have not been rigorous about presenting a framework for semantic interpretation. Now, we will propose to leverage the following ontological framework given by Liu (2003), which enumerates ways of resolving English into code:

- function $x.y()$
- ability $x.y()$
- param $x(y)$

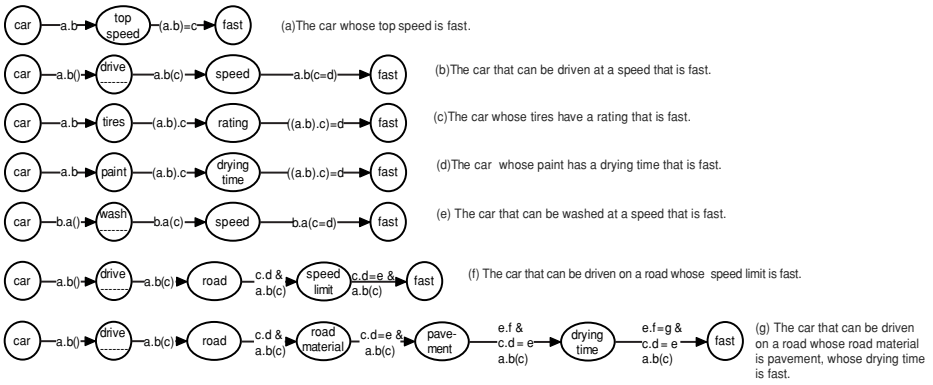

Figure 20.2. An underlying semantic model of English is used to generate interpretations of "fast car." From (Liu, 2003).

- property $x.y$
- isA $x{:}y$ (subtype)
- value $x = y$
- assoc x-related-to-y

In (Liu, 2003), it is proposed that natural language phrases can be understood in terms of compositions using the above ontology. An "interpretation" of a phrase is thus defined as one possible mapping from the surface language to some path in the network semantic model (Figure 20.2).

In our task of mapping surface natural language to programmatic code, we could view the problem in a way analogous to (Liu, 2003), i.e., an underlying semantic model of programming can be used to generate possible interpretations of inputted natural language, followed by the use of contextual cues, further semantic constraints, and dialog with the user to disambiguate from all possible interpretations to one or two likely interpretations.

Our approach to generating and selecting interpretive mappings from programmatic description to code is also supported by the natural language understanding literature, where there is precedent for exploiting semantic constraints for meaning disambiguation. BCL Spoken Language User Interface Toolkit (Liu, Alam and Hartomo, 2002), developed by BCL Technologies R&D, used Chomsky's Projection Principle and Parameters Model for command and control. In the principle and parameters model, surface features of natural language are seen as projections from the lexicon. The insight of this approach is that by explicitly parameterizing the possible behaviors of each lexical item, we can more easily perform language processing. We expect to be able to apply the principle and parameters model to our task, because the variables and structures present in computer programs can be seen as forming a naturally parameterized lexicon. An approach for using domain constraints to make natural language interaction reliable is also outlined in Yates (2003).

4.2. EVOLVABLE

The approach we have described thus far is fairly standard for natural language command-and-control systems. However, in our programming domain, the underlying semantic system is not static. Underlying objects can be created, used, and destroyed all within the breath of one sentence. This introduces the need for our language understanding system to be dynamic enough to *evolve* itself in real-time. The condition of the underlying semantic system including the state of objects and variables must be kept up-to-date and this model must be maximally exploited by all the modules of the language system for disambiguation. This is a challenge that is relatively uncommon to most language processing systems, in which the behavior of lexicons and grammars are usually defined or trained *a priori* and are not very amenable to change at run-time. Anyone who has endeavored to build a natural language programming system will likely have discovered that it is not simply the case that an off-the-shelf natural language processing packaging can be used.

To most optimally exploit the information given by the underlying semantic model, the natural language processing system will need to be intimately integrated with and informed by feedback from this evolving model. For example, consider the following fifth-grader non-programmer's description.

Pacman gets eaten if a ghost lands in the same space.

Without information from the underlying semantic model, some pretrained part-of-speech taggers will interpret "lands" as a noun, causing a cascade of misinterpretations, such as interpreting "ghost lands" as a new object. However, our underlying semantic model may know that "ghost" is a character in the game. If this knowledge is trickled back to the part-of-speech tagger, that tagger can have enough smarts to prefer the interpretation of "lands" as a verb untaken by the agent "ghost." This example illustrates that natural language processing must be intimately informed by the underlying semantic model, and ideally, the whole natural language programming system will be built end-to-end.

4.3. FLEXIBLE

Whereas traditional styles of language understanding consider every utterance to be relevant and therefore must be understood, we take the approach that in a "programming by talking" paradigm, some utterances are more salient than others. That is to say, we should take a selective parsing approach which resembles *information extraction*–style understanding. One criticism to this approach might be that it loses out on valuable information garnered from the user. However, we would argued that it is not necessary to fully understand every utterance in one pass because we are proposing a natural language dialog management system to further refine the information dictated by the user, giving the user more opportunities to fill in the gaps.

Such a strategy also pays off in its natural tolerance for user's disfluencies; thus, adding robustness to the understanding mechanism. In working with user's emails

in a natural language meeting command-and-control task, Liu et al. found that user disfluencies such as bad grammar, poor word choice, and run-on sentences deeply impacted the performance of traditional syntactic parsers based on fixed grammars. Liu et al. found better performance in a more flexible collocational semantic grammar, which spotted for certain words and phrases, while ignoring many less-important words which did not greatly affect semantic interpretation. The import of such an approach to our problem domain will be much greater robustness and a greater ability to handle unconstrained natural language.

4.4. ADAPTIVE

In working with any particular user in a programming task, it is desirable to recognize and exploit the specific discourse style of that user in order to increase the performance of the language understanding system. In our analysis of the natural language programming user studies performed by Pane and Myers (2004), we note that some users give a multi-tiered description of the program, starting with the most abstract description and iteratively becoming more concrete, while others proceed linearly and concretely in describing objects and functions. Consider for example, how the following two fifth-grader non-programmers begin their descriptions of Pacman quite differently:

The object of the game is to eat all the yellow dots. If you[‘re] corn[er]ed and there is a blinking dot eat that and the ghosts will turn a color and when you eat them you get 200 points. When you eat all the dots you win the game.

To tell the computer how to move the Pacman I would push letters, and arrows. If I push the letter A[,] pacman moves up. When I push Q it moves down. To make it go left and right I use the arrows.

Whereas the first non-programmer begins with a breadth-first description of the game, starting from the highest-level goals of the game, the second non-programmer begins with the behavioral specifics of user control, and never really explicates the overarching goals of game anywhere in the whole description. Understanding the descriptive style of the user allows us to improve the quality of the parsing and dialogue. If the user is accustomed to top–down multi-tiered descriptions like non-programmer #1, the system can assume that the first few utterances in a description will expose many of the globally salient objects in the semantic model that will later be referred to. For example, from the utterance, “*The object of the game is to eat all the yellow dots,*” we can assume that “yellow dots” are salient globally, and that “eat” is an action central to the game. If, however, the user is accustomed to giving details straight away like non-programmer #2, the system can perhaps be more proactive to ask the user for clarifications and context for what the program is about, e.g., asking the user, “Are you programming a game?”

There are also many other dimensions along with user style can vary, such as *inter alia*, example-driven scenario giving versus if-then-else explication, describing positive behavior of a system versus negative behavior, and giving first-person character description (e.g., “*You like to eat dots*”) versus third-person declarative description

(e.g., "*There is a Pacman who eats dots*") versus first-person user description (e.g., "*You press the left arrow key to move Pacman.*"). A natural language programming system should characterize and recognize many of these styles and style-dimensions, and to use this knowledge to inform both an adaptive case-based parsing strategy, and an adaptive case-based dialogue strategy.

4.5. CAN NOVICES' DESCRIPTIONS OF PROGRAMS BE FULLY OPERATIONALIZED?

In addition to concerns about natural language understanding *per se*, there is also the concern that novice descriptions of programs are vague, ambiguous, erroneous, and otherwise not fully precise in the way the programming language code would be. Our analysis of the CMU data shows that, indeed, this is often the case. But that does not render the cause of natural language programming hopeless. The imprecision manifests itself in different forms, each of which has important consequences for the dialog design.

4.6. INTENTIONAL DESCRIPTIONS

Above, we discussed some of the linguistic problems surrounding determining the referents of natural language expressions such as "it" or "him." These issues consist of figuring out ways to map expressions either to known objects in the program or to recognize when new objects are being introduced or created. In addition there is the problem of determining when objects are referred to by descriptive phrases rather than direct references.

We often saw descriptions such as "the player being chased," where in a conventional program, one might see a direct reference to a program variable. We need to be able to distinguish between intentional descriptions used to reference objects that the system knows about "at compile time," e.g., "When the ghosts chase a player, the player being chased has to run away" (two different ways of referring to a particular player), and those that imply a "run time" search, e.g., "Find a player who is being chased and turn him green."

Further, people use different levels of specificity to refer to an object. "Pac-Man" (by name), "the yellow circle" (by appearance), "the player" (by role) can be interchangeable in the discourse, but may have vastly different effects when the program is re-executed in a different context. In Programming by Example (Lieberman, 2001) this is referred to as the "data description problem," and is also a central problem here. The best way to deal with that problem is to give the user sufficient feedback when future examples are executed, so that the user will see the consequences of a learned description. New examples can be executed step-by-step, and the system can feed back its description, so that users understand the relation between descriptions and selected objects, and change them if they are incorrect. Systems like Lieberman, Nardi and Wright's Grammex (Lieberman et al., 2001) provide ways of incrementally generalizing and specializing descriptions or sub-descriptions so that a desired result is achieved. To some extent, however, the user needs to learn that the exact way in which an object is described

will have consequences for the learning system. Even at best, it is not possible to be as sloppy about descriptive phrases as we typically are in interpersonal discourse. This is a crucial part of what it means to learn to "think like a programmer."

4.7. ASSUMING CONTEXT

Natural language descriptions of programs tend to make a lot of assumptions about context that are not explicitly represented in the text. In programming languages, such context is represented by the scope of declarations and placement of statements. In natural language discourse, speakers assume that the listener can figure out what the context is, either by such cues as recently mentioned objects and actions, or by filling in necessary background knowledge. In doing natural language programming in the context of a programming by example system, we have the advantage of having a runtime context available which can help us discern what the user is talking about.

You should put the name and the score and move everyone below the new score down one.

Nowhere in this phrase does it include the implied context, "When one of the players has achieved a new total, and the scoreboard is displayed." However, if the user is programming interactively with concrete examples, the most likely time for the user to make such a statement is just *when* such a new score has occurred. It is the responsibility of the programming environment to figure out that what is important about the current situation is the posting of a new score. In Wolber and Myers (2001) discuss the problem of demonstrating *when* to do something as well as *how* to do it, under the rubric of Stimulus-Response Programming by Example.

Again, in the case of failure to recognize the context for a statement, the strategy is to initiate a dialogue with the user explicitly about in what context the statement is to be taken.

4.8. OUT-OF-ORDER SEQUENCES AND ADVICE

As noted by Pane and Myers (2004), users tend not to think linearly, and provide instructions that are not properly ordered, which is why they adopted an event driven style for HANDS. Sometimes, this is a matter of making assumptions about the temporal context in which commands will be executed.

Packman gets eaten if a ghost lands in the same space as Packman.
If Packman gets a power pill, then he gets points by landing in the same space as a ghost.

Taken literally as code, these statements are in the wrong order—the condition of eating the power pill should be checked *before* deciding what action to take when Pac-Man and the ghost arrive at the same spot. But the user is adopting the quite reasonable strategy of telling us what the *usual* or most common case is first, and only then informing us about the rare exceptions. The system needs to be able to untangle such cases.

Other natural language statements provide *advice*. Advice is not directly executable, but may affect what gets executed at future points in the interaction. Advice may incrementally supply parameters or modifiers to other commands. Advice may affect the level of generality of object descriptions. This is an important style of interaction that is not well supported by current programming methodologies (Lieberman, 2001).

The object of the game is to eat as many dots as you can without getting eaten by the ghosts.

Some utterances are not actually code themselves, but directives to *make edits* to the program.

When monsters are red . . . [they] run . . . to the other side of the screen. Same goes for Pac-Man.

Here, "same goes" means "write the same code for Pac-Man as you did for the red monsters." This suggests that users are assuming a capability for high level program manipulation, that can, for example, insert statements into already-written code or generate code from design patterns. The best-known project for providing such capabilities directly to a programmer is The Programmer's Apprentice (Rich, 1990).

4.9. MISSING OR CONFLICTING CASES

Because order constraints are more relaxed in a natural language style interaction, it is often more difficult to determine if all cases have been covered. Of course, even in conventional programming, nothing prevents writing underconstrained or overconstrained programs. Some software engineering test methodologies for end users do attempt to infer case coverage in some situations (Ruthruff), and we envision similar techniques might be applied in our domain.

In an event driven style as advocated by Pane and Myers it is also possible for handlers of different events to conflict. Graphical- and example-based feedback helps avoid, and catch cases of, underconstrained and overconstrained situations. We also like the critique-based interaction found by McDaniel (2001), where directions "stop this" (for overconstrained situations) and "do something" (for underconstrained situations) correct the system's responses.

4.10. CHANGE OF PERSPECTIVE

Users do not always describe things from a consistent viewpoint. They may switch, unannounced, between local and global viewpoints, between subjective and objective viewpoints, between the viewpoints of various actors in the program. For example, a user might say *"When you hit a wall . . . "* (*you* meaning a screen representation of a game character), and *"When you score a point . . . "* (*you* meaning the human user), in the same program without warning. Again, this is a form of missing context which

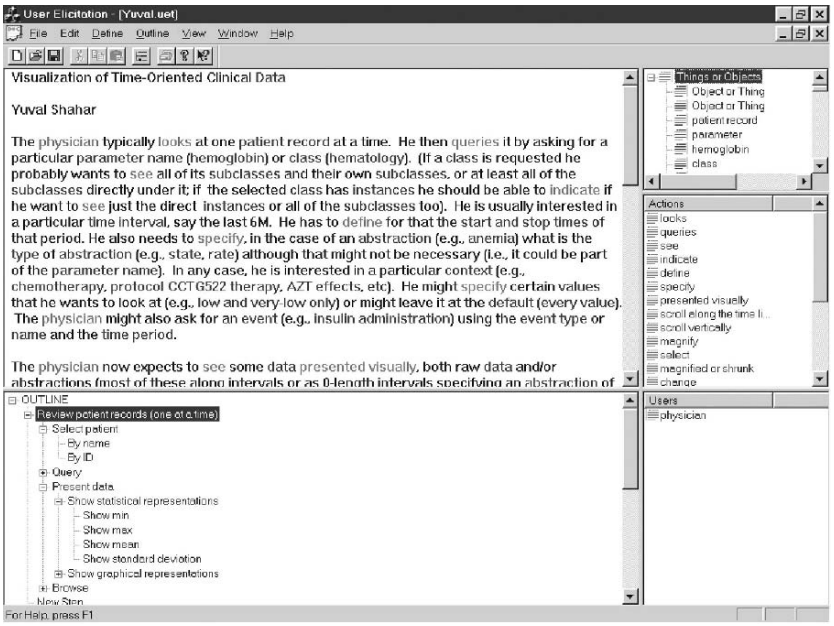

Figure 20.3. Tam et al's U-Tel lets users annotate text describing a program.

the reader is expected to supply. People do recognize the inherent ambiguity of such references, so they are often willing to supply clarification if necessary.

5. Annotation Interfaces

Another, less radical, possibility for a natural language programming interface is to let the user *annotate* a natural language description. The idea would be for the user to type or speech-transcribe a natural language description of the program, and then manually select pieces of the text that correspond to meaningful entities in the program. This reduces the burden on the system of reliably parsing the text. Such an approach was taken by Tam et al., (1998) in U-Tel (see Figure 20.3), which has an underlying model-based interface methodology. In the illustration, the full text appears in the upper left, and highlighted words for "steps," "actions," and "objects" are collected in the other panes.

U-Tel, however, did not construct a complete program; rather it functioned mainly as a knowledge elicitation aid, and required further action by a programmer conversant in the Mobi-D model-based formalism.

The annotation approach is attractive in many circumstances, particularly where a natural language description of a procedure already exists, perhaps for the purpose of communicating the procedure to other people. We believe the approach could be extended to support full procedural programming. Other attractive features of this approach are that it is less sensitive to order, and does not require the system to

understand everything the user says. Even in natural language programming, users "comment" their code!

6. Note

Portions of this paper were written by dictation into the speech recognition program IBM ViaVoice, by the first author when he was recovering from hand injuries sustained in a bicycle accident.

Current speech interfaces are not good enough to perform unaided transcription; all require a mixed-initiative critique and correction interface to display recognition hypotheses and allow rapid correction. The author thus experienced many issues similar to those that will arise in natural language programming; among them: inherent ambiguity (no speech program can distinguish between *too* and *two* by audio alone), underspecification and misunderstanding of natural language directives. Although today's speech interfaces leave a lot to be desired, we were struck by how successfully the interaction is able to make up for deficiencies in the underlying recognition; this gives us hope for the approach. We apologize for any errors that remain in the paper as a result of the transcription.

7. Conclusion

Programming directly in natural language, without the need for a formal programming language, has long been a dream of computer science. Even COBOL, one of the very early programming languages, and for a long time, the dominant business programming language, was designed to look as close as possible to natural language to enhance readability. Since then, very few have explored the possibility that natural language programming could be made to work.

In this paper, we have proposed an approach based on advances in natural language parsing technology, mixed-initiative dialog, and programming by example. To assess the feasibility of such an approach we have analyzed dialogs taken from experiments where non-programmer users were asked to describe tasks, and it seems that many of the important features of these dialogs can be handled by this approach. We look forward to the day when computers will do what we say, if only we ask them nicely.

Acknowledgments

We would like to thank John Pane and Brad Myers for sharing with us the data for their Natural Programming experiments.

References

Alam, H., Rahman, A., Tjahjadi, T., H. Cheng, H., Llido, P., Kumar, A., Hartono, R., Tarnikova, Y. and Wilcox, C. (2002). Development of spoken language user interfaces: A tool kit approach. In:

T. Caelli, A. Amin, R. Duin, M. Kamel and D. Ridder (eds.), *Lecture Notes in Computer Science LNCS 2396*, 339–347.

Collins, M. (2003). Head-driven statistical models for natural language parsing. *Computational Linguistics* **29**(4), MIT Press, Cambridge, MA, USA.

Lieberman, H., Nardi, B. and Wright, D. (2001). Training agents to recognize text by example. In: H. Lieberman (ed.), *Your Wish is My Command: Programming by Example*, Morgan Kaufmann, San Francisco, CA, USA.

Lieberman, H. (2001). Interfaces that give and take advice, in human–computer interaction for the new millenium, John Carroll (ed.), ACM Press/Addison-Wesley, Reading, MA, USA, pp. 475–485.

Lieberman, H. and Liu, H. (2002). Adaptive linking between text and photos using common sense reasoning. In: De Bra, Brusilovsky, Conejo (eds.), *Adaptive Hypermedia and Adaptive Web*, 2nd International Conference, AH 2002, Malaga, Spain, May 29–31, Proceedings. Lecture Notes in Computer Science 2347 Springer 2002, ISBN 3-540-43737-1, pp. 2–11.

Liu, H., Alam, H. and Hartono, R. Meeting Runner: An Automatic Email-Based Meeting Scheduler. BCL Technologies—US. Dept of Commerce ATP Contract Technical Report. Available at: http://www.media.mit.edu/∼hugo/publications/

Liu, H. (2003). Unpacking meaning from words: A context-centered approach to computational lexicon design. In: Blackburn et al. (eds.), Modeling and Using Context, 4th International and Interdisciplinary Conference, CONTEXT 2003, Stanford, CA, USA, June 23–25, 2003, Proceedings. Lecture Notes in Computer Science 2680 Springer 2003, ISBN 3-540-40380-9, pp. 218–232.

Liu, H. (2003). MontyLingua: An End-to-End Understander for English. At: http://web.media.mit.edu/∼hugo/montylingua.

McDaniel, R. (2001). Demonstrating the Hidden Features That Make an Application Work, in Your Wish is My Command, H. Lieberman (ed.), Morgan Kaufmann, San Francisco, CA, USA.

Pane, J.F., Ratanamahatana, C.A. and Myers, B.A. (2001). Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, **54**(2), 237–264. http://web.cs.cmu.edu/∼pane/IJHCS.html.

Pane, J.F. and Myers B.A. (2004). More Natural Programming Languages and Environments, in End-User Development, Kluwer, Dordrecht, Netherlands.

Rich, C., Sidner, C.L. and Lesh, N.B. (2001). COLLAGEN: Applying collaborative discourse theory to human–computer interaction. *Artificial Intelligence*, **22**(4), Winter, 15–25.

Rich, C.H. and Waters, R.C. (1990). *The Programmer's Apprentice*. Reading, MA, USA: Addison-Wesley.

Ruthruff, J.R., Prabhakararao, S., Reichwein, J., Cook, C., Creswick, E. and Burnett, M. (in appear). Interactive Visual Fault Localization Support for End-User Programmers, *Journal of Visual Languages and Computing*.

Stocky, T., Faaborg, A., Espinosa, J., Lieberman, H. A Commonsense Approch to Predictive Text Entry, ACM Conference on Computer-Human Interaction (CHI-04), Vienna, Austria, April 2004.

Tam, R.C., Maulsby, D. and Puerta, A.R. U-TEL: A Tool for Eliciting User Task Models from Domain Experts. IUI98: International Conference on Intelligent User Interfaces, San Francisco, January 1998, pp. 77–80.

Wolber, D. and Myers, B. (2001). Stimulus-Response PBD: Demontrating "When" as Well as "What", In: H. Lieberman (ed.), *Your Wish is My Command*, Morgan Kaufmann, San Francisco, CA, USA.

Yates, A., Etzioni O. and Weld, D. (2003). A Reliable Natural Language Interface to Household Appliances, Conference on Intelligent User Interfaces, Miami, Florida. *Association for Computing Machinery*, New York, NY, USA.