GHOST in the Machine

Being a recounting of one man's experience in learning all about the cognitive architecture ACT-R, by trying to teach it how to play a silly word game.

Steven Matuszek
CMSC 691M
Spring 2003
UMBC

Contents:

Introduction to the problem domain

[Scene: Guildenfoo and Rosenbar are loitering about the castle grounds, playing GHOST.]

Rosenbar: My turn to start. P.

Guildenfoo: P E.

Rosenbar: P E R.

Guildenfoo: P E R V.

Rosenbar: P E R V I.

Guildenfoo: I? *[mutters to himself for forty seconds]* I Challenge.

Rosenbar: PERVICACIOUS.

Guildenfoo: Oh, of course.

The play of the word game GHOST is to take turns naming letters that build onto a word. You lose if you complete a word (of more than three letters), so you must attempt to manoeuvre your opponent or opponents into this situation while avoiding it yourself. Rosenbar's word PERVICACIOUS would have ended with Guildenfoo supplying the final S, so it was a good word for his purposes.

Guildenfoo was thinking of the word PERVERSE, which would have ended on Rosenbar. When Rosenbar supplied an I instead, Guildenfoo could not think of any word that started that way, so he exercised his right to Challenge instead. If Rosenbar did not have a valid word—if he was bluffing, or if his word was misspelled, a proper noun, or otherwise unacceptable—he would have lost the round. However, since he did have a word, Guildenfoo lost the round.

It is clear to see that a computer program could be very, very good at this game indeed. With the ability to store hundreds of thousands of words, recall them instantly, and even search them to find words that are sure to end on its opponent, such an agent would be no fun to play against whatsoever.

But what if we had an agent that couldn't always remember words? That ran into the same difficulties dredging its memory for appropriate words, no matter how pervicaciously it tried? That had more trouble coming up with seldom-used words than common ones? That didn't have instant recall, but rather was subject to memory latency much like that of a human being?

Introduction to the architecture

ACT-R is "a production system theory that tries to explain human cognition by developing a model of the knowledge structures that underlie cognition." [Bothell]

ACT-R 5 is the latest in a long line of cognitive architectures based on the theories of John Anderson of Carnegie Mellon University. Its history cannot be done justice in this space; more information is available at http://act-r.psy.cmu.edu/.

The essential elements of the ACT-R model of human cognition are

- *chunks*, which are facts that it remembers,
- *productions*, which are rules or patterns of behavior that it follows, and
- *buffers,* including buffers for
    - *retrieval*, which stores the chunk that the mind has most recently fetched from its declarative memory,
    - *goal*, which stores the task the mind is currently trying to accomplish, and
    - *visual, manual* and *aural* buffers, which simulate sight, motor activity, and hearing in the brain.

ACT-R researchers claim that these actually correspond to specific structures in the human brain and the ways in which they operate, and that therefore this architecture has a high degree of cognitive plausibility.

ACT-R uses a numeric measure to decide among the chunks that it is asked to retrieve, and to decide among the productions that seem to be relevant to the current situation. This measure is called *activation* for chunks and *utility* for productions. The values of these measures are affected by how recently they have been used, how frequently they have been used, exactly how similar they are to what is sought, and even a certain degree of stochastic noise. All this is, again, said to be consistent with neurological studies.

We are not neurologists, though. Can this help us make a system that plays GHOST, and is fallible in human ways?

This project proposes, in part, to show that it can. If nothing else, activation levels should be useful to make the system not be able to come up with a word that it knows, or even accidentally come up with words that don't correctly match the spelling.

Issues and implementation on the front end

ACT-R is written in LISP. Thus, a program that makes use of it should be written in LISP, and call the functions that it exposes. However, cognitive plausibility is to be kept in mind, and the designers did not expose many functions that would allow one to reach in and directly mess with the contents of the goal and retrieval buffers and such things.

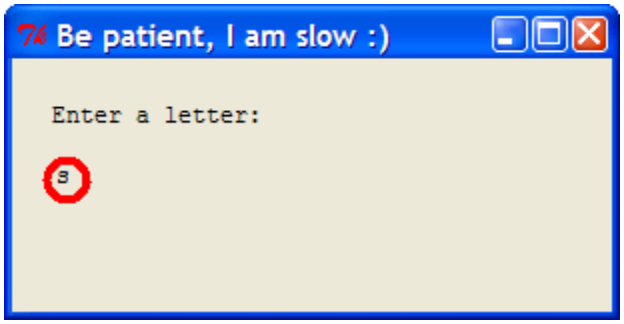
Visual perception and motor activity

Rather, the ideal LISP program should interact with the ACT-R model by showing it things using its visual buffer or telling it things using its aural buffer. The model would then respond by using its manual capabilities to type keys in response.

I set out to create such an interface. The following program

- displays the words `Enter a letter`:,
- has the visual system look at the screen and note all the words, so that it will ignore them and focus on the new letter when the user types it in,
- displays the new letter when the user types it in,
- uses the visual system to read it in and recognizes it as a letter,
- uses the manual system to "type" a letter in response.

Here is a screenshot of the program right after it has read in the user's letter. The red circle is an optional feature that shows the visual system's current point of focus.



The ACT-R productions that create this behavior look like this:

```
(P start
      =goal>
            ISA look-for-word
            status nothing
==>
      +visual-location>
            ISA visual-location
            attended nil
      =goal>
            status looking
)
```

This first production says that if our goal is to look for a word, and we have nothing so far, then find a visual location on the screen that we have not yet paid attention to and put it into the `visual-location` buffer. Also, let the `goal` buffer record that we are looking.

```
(P attend-letter
      =goal>
            ISA look-for-word
            status looking
      =visual-location>
            ISA visual-location
      =visual-state>
            ISA module-state
            modality free
==>
      +visual>
            ISA visual-object
            screen-pos =visual-location
      =goal>
            status reading
)
```

This second production specifies that if our `goal` is to look for a word, and our status is looking—so this should follow from the results of the first production—then we should use the results of the `visual-location` buffer to put something into the `visual` buffer (in other words, take a look at what we found).

```
(P encode-letter
      =goal>
            ISA look-for-word
            status reading
      =visual>
            ISA text
            value =whatIfound
==>
      =goal>
            result =whatIfound
            status done
      +manual>
            isa press-key
            key =z
)
```

The last production says that if a text object was retrieved and stored in the `visual` buffer, we should store the `value` of the object into the `result` slot of the `goal` buffer, and also tell the `manual` buffer to "press" the `z` key. (When a key is pressed, whether physically by the user or programmatically by the model, a LISP event handler is triggered.)

Now all I would have to do is to combine my productions for visual and motor interactions with my productions for playing GHOST.

As a programmer, however, I did not want to do that.

I could not divine the mechanisms by which I could get a reasonable amount of abstraction between the user interface and the domain logic. It is certainly the case that many human processes have very fuzzy layers of abstraction, but I think that in the case of GHOST specifically, the patterns for what letter to say next have very little to do with saying it out loud.

Dynamic goal creation

So, I punted.

Rather than having the model read from the window and combine the results with its game-playing goals, I decided to let the outer LISP program handle the user input events, and generate goals for the model to strive for based on that input.

The meat of this logic looks like this:

```
(setq newgoal (list 'status 'waiting))
(when (>= *length-so-far* 1)
   (setq newgoal (append newgoal (list 'st1 (first *word-so-far*)))))
(when (>= *length-so-far* 2)
   (setq newgoal (append newgoal (list 'st2 (second *word-so-far*)))))
(when (>= *length-so-far* 3)
   (setq newgoal (append newgoal (list 'st3 (third *word-so-far*)))))
(when (>= *length-so-far* 4)
   (setq newgoal (append newgoal (list 'st4 (fourth *word-so-far*)))))
(when (>= *length-so-far* 5)
   (setq newgoal (append newgoal (list 'st5 (fifth *word-so-far*)))))
(when (>= *length-so-far* 6)
   (setq newgoal (append newgoal (list 'st6 (sixth *word-so-far*)))))
(when (>= *length-so-far* 7)
   (setq newgoal (append newgoal (list 'st7 (seventh *word-so-far*)))))
(when (>= *length-so-far* 8)
   (setq newgoal (append newgoal (list 'st8 (eighth *word-so-far*)))))
(when (>= *length-so-far* 9)
   (setq newgoal (append newgoal (list 'st9 (ninth *word-so-far*)))))
(setq newgoal (append
   (list newgoalname 'isa 'find-word-starting-with) newgoal))

;  Add this goal to our declarative memory
(add-dm-fct (list newgoal))

;  and focus on it.
(goal-focus-fct (list newgoalname))
```

If the list `*word-so-far*` is currently `("t" "e" "l" "e")`, this code will generate the new goal as `(ghost-goal44 isa find-word-starting-with st1 "t" st2 "e" st3 "l" st4 "e" status waiting)`. It will then add it to the model's declarative memory, and ask the model to focus on solving this goal.

Is this cheating? I prefer to think of it as deliberation.

The model still communicates by "typing" into the window. ACT-R exposes functions to insert new goals and new chunks into declarative memory, but unless I am missing something, it does not expose methods to, *e.g.*, retrieve the current contents of the retrieval buffer.

Fortunately, having it type into the window is easy when all we need to have it do is say a letter at a time. I would not wish to attempt to communicate anything more complicated in this manner.

Issues and implementation on the back end

Basic chunks and productions

Here's how my model stores word spellings in declarative memory.

```
(chunk-type word le1 le2 le3 le4 le5 le6 le7 le8 le9 le10)

(w-apple ISA word  le1 "a" le2 "p" le3 "p" le4 "l" le5 "e")
(w-bear ISA word  le1 "b" le2 "e" le3 "a" le4 "r")
(w-catch ISA word  le1 "c" le2 "a" le3 "t" le4 "c" le5 "h")
(w-dogged ISA word  le1 "d" le2 "o" le3 "g" le4 "g" le5 "e" le6 "d")
(w-elephant ISA word
      le1 "e" le2 "l" le3 "e" le4 "p" le5 "h" le6 "a" le7 "n" le8 "t")
(w-fish ISA word  le1 "f" le2 "i" le3 "s" le4 "h")
 ...
```

The `chunk-type` statement specifies that any chunk that `ISA word` has ten named slots, each of which can store a value. Not all the slots will necessarily be filled.

Each individual word chunk has a name (such as `w-fish`), a specification that it `ISA word`, and a letter in each slot, spelling the word out. The requirement that these slots hold letters is only semantic, not syntactic.

Here is a sample goal, which will look for a word that begins with `GI`.

```
(goal-1 ISA find-word-starting-with st1 "g" st2 "i" status waiting)
```

And here are the two productions which, the system will determine, match this goal.

```
(p irol2
      =goal>
            isa find-word-starting-with
            st1 =alpha
            st2 =beta
            st3 nil
            status waiting
==>
      =goal>
            status retrieving
      +retrieval>
            isa word
            le1 =alpha
            le2 =beta
)
```

This first production will match the fact that the `goal` is a `find-word-starting-with` whose `status` is `waiting`. It will bind the variable `alpha` to the `st1` slot of the goal, which in this case is `"g"`. It will bind the variable `beta` to the `st2` slot of the goal, which in this case is `"i"`. It is critical that the `st3` slot be `nil` (unspecified) in the goal, because this delineates that we are searching on exactly the first two letters—no more, no fewer.

The first action that this production will take is to set the `status` of the `goal` to `retrieving`. This will allow the production-matcher to move on to the second production, shown below.

The second action taken by the first production is to request a retrieval. It is asking the model to find a chunk that `ISA word`, and that has an `le1` slot whose value is `alpha` and an `le2` slot whose value is `beta`.

In other words, it will search through the declarative memory, which contains all the spellings-of-words chunks we specified above. It will examine each one to see whether the first two letters match. The one that it decides to return—and we explain that decision process later, when we introduce uncertainty into it—will be loaded into the `retrieval` buffer.

```
(p crol2
      =goal>
            isa find-word-starting-with
            st1 =alpha
            st2 =beta
            st3 nil
            status retrieving
      =retrieval>
            isa word
            le3 =lastletter
==>
      =goal>
            status found
      +manual>
            isa press-key
            key =lastletter
)
```

This second production should always fire after the first one does—note that the only difference between the conditions, which is everything before the `==>`, is that the `status` has changed from `waiting` to `retrieving`, and this change is effected by the first production.

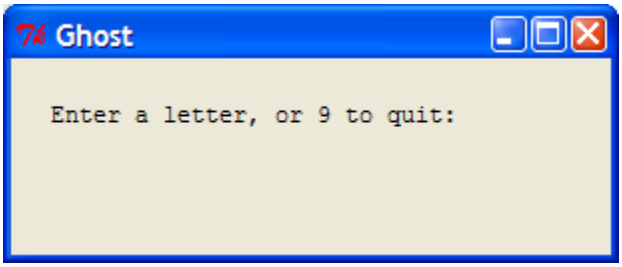
Or rather, that is the only difference between the `=goal>` parts of the two productions. The second production introduces a `=retrieval>` element, which matches on the contents of the retrieval buffer. If the first production's request was successfully granted, the retrieval buffer should store a chunk now, a chunk such as

```
(w-giblet ISA word  le1 "g" le2 "i" le3 "b" le4 "l" le5 "e" le6 "t").
```

We now bind the variable lastletter to the third letter of the retrieved word, whatever it is.

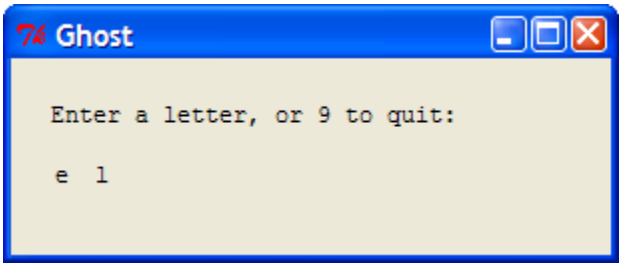
The right-hand side of the production can now use the manual buffer to press the key for the last letter—this is how the model takes its turn and says a letter!

Let's see the model in action.



It is the user's turn first.

The user presses E.



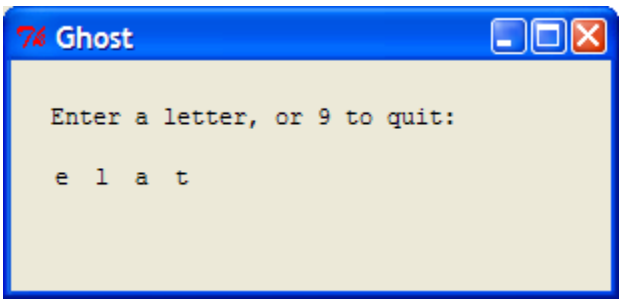
The model immediately—immediately, since there are not that many words in its corpus at the moment, matches on the word ELEPHANT. Here is a trace of its productions:

```
Time  0.000: Initiate-Retrieval-On-First-Letter Selected
 Time  0.050: Initiate-Retrieval-On-First-Letter Fired
 Time  0.050: W-Elephant Retrieved
 Time  0.050: Continue-Retrieval-On-First-Letter Selected
 Time  0.100: Continue-Retrieval-On-First-Letter Fired
 Time  0.100: Module :MOTOR running command PRESS-KEY
 Time  0.250: Module :MOTOR running command PREPARATION-COMPLETE
 Time  0.300: Module :MOTOR running command INITIATION-COMPLETE
 Time  0.310: Device running command OUTPUT-KEY
```

It fires the first production. The chunk `W-Elephant` is retrieved. It fires the second production, and uses its manual/motor module to "press" L.

`Initiate-Retrieval-On-Letter`, incidentally, is what `irol` and `crol` stand for in the production names above.

The user then enters A.



The model this time finds the chunk `W-Elation`, and responds with a T.

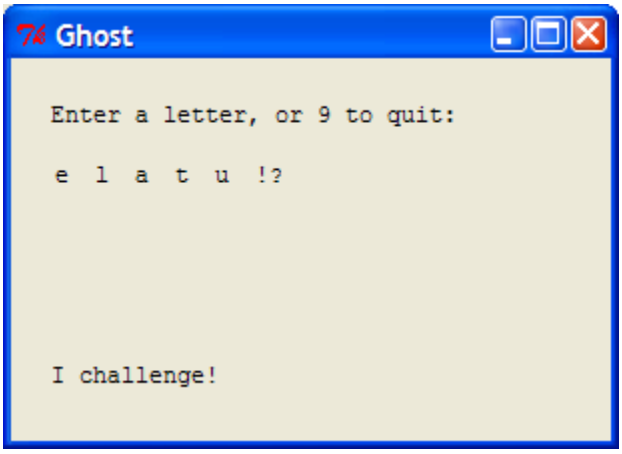
The user presses U.

```
Time  1.050: Irol5 Selected
 Time  1.100: Irol5 Fired
 Time  1.100: Failure Retrieved
```

Oh dear, the model couldn't find any chunks that start with E L A T U. Now what happens? Enter a new production rule:

```
(p challenge
      =goal>
            isa find-word-starting-with
            status retrieving
      =retrieval>
            isa ERROR
            condition Failure
==>
      =goal>
            status challenge
      +manual>
            isa press-key
            key "8"
)
```

This production checks for the condition in which the retrieval process could not find any matching chunks. In this case, we want the GHOST model to Challenge its opponent. It does so by pressing the 8 key, which the LISP program interprets as a Challenge.



And that is where this game ends. (The program does not currently allow the user to respond to a challenge. Nor, for that matter, does it notice when the user or the model has inadvertently completed a word. These are relatively cosmetic issues.)

Cognitive plausibility refinements

This is all well and good, but what about the promise of ACT-R to think more like a human being?

Recall that productions in ACT-R can have *utility* values that affect how often they are selected to be executed. We are not concerned with those at the moment, because the only productions we are using are the basic rules of the game. If we were to add productions that implement strategy, we would definitely be interested in this aspect.

Recall also that chunks in ACT-R can have *activation* values that affect how often they are retrieved on a retrieval request. We are definitely interested in that. This section will address the components of the activation values, and how we can use them to improve the cognitive plausibility of our system.

Noise

ACT-R permits two different kinds of noise that affect the activation levels of individual chunks. One, *permanent activation noise*, is stochastically generated for each chunk when it is first created, and the other, *activation noise*, is stochastically generated at each retrieval attempt. (The exact equations involve exponential decay and, again, are ostensibly backed up by biology).

Here are two activation traces from a production that seeks to match a word whose first letter is `Q` and second letter is `U`, in a corpus containing `W-Queen` and `W-Quit`:

```
   Adding noise  0.175
Partial matching chunk W-Queen with activation  0.140
Similarity between chunks "q" and "q" is  0.800
Adjusting activation by  0.800 to  0.940
Similarity between chunks "u" and "u" is  0.800
Adjusting activation by  0.800 to  1.740
Matching score of chunk W-Queen is  1.740.
Activation  1.740 is larger than previous best -0.500: selecting W-Queen.

   Adding noise  0.538
Partial matching chunk W-Quit with activation  0.544
Similarity between chunks "q" and "q" is  0.800
Adjusting activation by  0.800 to  1.344
Similarity between chunks "u" and "u" is  0.800
Adjusting activation by  0.800 to  2.144
Matching score of chunk W-Quit is  2.144.
Activation  2.144 is larger than previous best  1.740: selecting W-Quit.
```

First, please note the last line of each block. This makes clear that the model examines every chunk, and selects the one whose activation level is highest.

Second, note that the per-retrieval noise is quoted on the first line of each chunk.

You can see that the only difference between these two is the noise. This is desirable for having it select different words at random. The range of the noise is a parameter that can be set, and adjusted relative to the other elements that affect the activation level.

Forgetting

ACT-R permits you to set a value called the *retrieval threshold*, which is the minimum activation level that a chunk must reach in order to be retrieved.

In the previous example, if the retrieval threshold had been set to 2.0, `W-Quit` would have cleared the threshold, but `W-Queen` would not have. In other words, the model would have been unable to think of the word QUEEN.

Is this likely for a word such as QUEEN? Not really. But how about a word such as PERVICACIOUS? Unfortunately, the only difference between activation levels we've seen so far is random...

Learning

What we'd really like is for words that get used more often than other words to have higher activation values. This can be done in the ACT-R model by turning on its learning parameter.

To test this, I set up a GHOST system which knew only a few words starting with QU, and only one starting with QUA. Then I had it run eight goals that all wanted a word starting with QUA.

Thus, the same chunk was selected for retrieval, and used in another production, eright times. On the last run through, ACT-R reported the following sources of activation levels:

```
Sources of activation are: (Dummy7 Retrieving)
 Computing a base level of  0.498 from 1 references from creation time  0.000
 Computing a base level of  0.498 from 1 references from creation time  0.000
 Computing a base level of  0.498 from 1 references from creation time  0.000
 Computing a base level of  2.578 from 8 references from creation time  0.000
 Computing a base level of  0.498 from 1 references from creation time  0.000
 CHUNK W-Quarry Activation  2.578 Latency  0.076
```

The activation levels of `0.498` belonged to the other words, and the activation level of `2.578` belonged to `W-Quarry`. Its activation level was increased with every use. In time, it will go down again, according to the designers' decay equations.

If our pet brain, besides playing GHOST, had to actually read or converse, then more common words would end up with much higher activation levels. This is very much what we'd like to happen, but is outside the scope of our experiments here. But it would not be outré to expect the model to become especially fond of words like FJORD if it played Ghost a few thousand times.

Latency

There is also a parameter for activation latency. If we were to set this parameter, it would result in those words that have lower activation (say, because they're seldom used) taking longer for the model to summon.

More importantly, ACT-R constrains every retrieval attempt to take a certain minimum (and plausible) amount of time. This means that our model will have to sit and think for quite some time, rather than binary-searching through its array of all valid words.

Misspelling

Recall this line from the activation trace:

```
Similarity between chunks "q" and "q" is  0.800
```

The `0.800` value is a settable parameter for how similar a chunk is to itself. It is also possible to explicitly set the similarities of specific pairs of chunks. For example, if I were to set the similarity of `Letter-C-chunk` and `Letter-S-chunk` to `0.500`, then there would be a certain chance that a search for a word starting `R E S E` would return `W-Receive`.

I did not attempt this experiment because, currently, spellings are not implemented with `Letter-C-chunk`s, but with string constants.

Also, it's not a very sophisticated model of spelling errors.

Conclusions

Observations on the architecture

One of the purposes of ACT-R is to simulate human performance for purposes such as testing educational software [Lebiere].

I would have to get much more deeply into productions before I could speak to it on that score, but my experiences with its model of declarative memory recall are encouraging. I was often surprised by how often it seemed to be sitting there thinking things over before it responded to a request.

Also, the visual model seemed quite plausible in the order it "looked" at things on the screen, the speed at which it did so, and its ability to remember (without declarative memory) what it had already seen, and notice what was new.

Writing an outer program that could only communicate with the model in limited ways was a trial, but I am convinced that it makes sense for purposes such as testing other programs, and for reasons of abstraction.

I could wish that it were better documented; as a development environment, it is clearly not exactly ready for prime time. The tutorials were well written, but I shouldn't have had to guess at the syntax of the commands.

I could use a tutorial on how to stack up related goals.

And it needs a way to kill infinite loops!

Simplifying compromises made and future work

There are several compromises that were made for development purposes, some of which would be easy to improve upon and some of which would not.

Cosmetic gameplay issues

These include letting the computer go first, letting the user challenge words, and having the program notice when a word has been completed. Some of these would involve some changes to the model, and some only to the program.

One interesting question is: should the program (as opposed to the model) have a list of all actual words and their correct spellings? Or should the question of whether something is a good word or not fall to the model, as it does to the human players in real life?

A bigger and more realistic corpus

Right now, the declarative memory does not know many words (about 860).

I could easily add more, but this is for the purpose of getting the system working. The average native speaker of a language uses on the order of tens of thousands of words, and can verify as words many that she could not come up with if asked to.

The way the system is set up, it examines every single word chunk in its declarative memory, which is obviously not what humans do! I suspect that humans have them hashed by several aspects. One scheme would be to hash them by first letter. That is, we would have chunks like `(W-slam ISA s-word le1 "s" le2 "l" le3 "a" le4 "m")`. I think this both is cognitively plausible and would result in better performance by a factor of about 13.

When we were generating the chunks from the word list (I wrote a Java program to do this), we could also give each word a starting activation level based on its frequency of use in real life.

The corpus also currently purposely omits words that contain other words as prefixes. This should be handled by a production instead.

A better (mis)spelling model

What's more, if I gave a human the three letters G H O, and asked for a word that began with them, the human would almost certainly sit there saying "Go…. ga… gose… gahm…" That is to say, much word retrieval must surely be done based on pronunciation.

Thus, swapping letters for other letters is an overly simplistic model of misspelling words.

No doubt there has been research done into the actual mental models of spelling and pronouncing words, that we could follow in order to have a more realistic model of spelling and misspelling words.

However, in the course of this project, reading research took a back seat to experimental research. I yet intend to look into this to some extent.

It doesn't try to win!

All the model currently does is try to keep building valid words. It doesn't care who would say the last letter. There would be a very simple fix for this: store the parity of each word as part of its chunk.

But this is not cognitively plausible. The way humans actually do this—or the way I do, at any rate—is to think, "B her, R me, E her, A me..."

This would be a complicated set of productions, but should be feasible.

Other things a smarter model might try to do to win:

- Remember words that end on it, and try one of those in the hopes that the other person won't be able to think of it and will have to Challenge.
- For the letter it is about to say, check to make sure that there are no obvious words that begin with those letters and would end on it.
- Try a common letter like A, and act smug.
- Try some different forms of bad words (for example, an adjective ending in –Y has a form –IEST of opposite "badness" parity).
- Try to get at certain words that are devastating winners. Most people will challenge you on FJ. There are also lots of dodgy words that have a Q not followed by U (just ask a Scrabble player). This one could be achieved with starting activation levels.

Final conclusion

I believe I have succeeded in my stated goal of getting ACT-R to play GHOST very badly.

References

Rules of GHOST: http://www.kith.org/logos/words/lower/g.html

Dan J. Bothell, ACT-R 5.0 Tutorial Units 1, 2, 3, 5, 6, 7, http://act-r.psy.cmu.edu/tutorials/

Dan J. Bothell, Introduction to the ACT-R GUI Interface,
http://act-r.psy.cmu.edu/tutorials/AGI.html

Christian Lebiere, Introduction to ACT-R,
http://act-r.psy.cmu.edu/tutorials/ACT-R_intro_tutorial.ppt

Unknown ACT-R developer, some code I looked at to see syntax,
http://act.psy.cmu.edu/models/fmri_experiment2.model