Making it Real:
Loebner-winning Chatbot Design

by
Bruce Wilcox, B.A. Comp.Sci.
Sue Wilcox, B.A. Psych., M.A. Soc.Admin., M.Sc. Comp.Sci., Dipl. Fine Arts
gowilcox@gmail.com
2500 Kalakaua Avenue #2204
Honolulu, HI 96815

Summary

For the last three years, our chatbots have come in 1st twice and 2nd once in the Loebner Prize Contest, with a different persona each year (Suzette, Rosette, Angela). Suzette even fooled a human judge.

A world-class chatbot should tell the story of its life, have a consistent personality, and respond emotionally. It takes a lot of script. And it takes a powerful engine designed to support natural language processing in a variety of ways and make it relatively easy to author all that script.

This paper briefly discusses ChatScript, the open-source Natural Language scripting language and engine running our bots. Then it looks at how we construct chatbots and what we have learned.

Resumen

Durante los últimos tres años, nuestros chatbots han conseguido dos veces el primer puesto y una el segundo en el concurso Premio Loebner, con un personaje distinto cada año (Suzette, Rosette, Angela). Suzette, incluso consiguió engañar a un juez humano.

Un chatbot de clase mundial debe contar la historia de su vida, tener una personalidad coherente y responder emocionalmente. Se necesita una gran cantidad de secuencias de comandos así como un motor de gran alcance diseñado para apoyar el procesamiento del lenguaje natural en diferentes maneras y lograr que le resulte relativamente fácil al autor toda esa secuencia de comandos.

En este artículo se analiza brevemente ChatScript, el código abierto Natural Language en el que programar y poner en marcha nuestros robots. A continuación, se describe como construimos chatbots y lo que hemos aprendido hasta ahora.

Keywords

chat, chatbot, ChatScript, conversation, Loebner, Artificial Intelligence, Natural Language

Palabras clave

chat, chatbot, ChatScript, conversación, Loebner, Inteligencia Artificial, Lenguaje Natural

Introduction

The task of a chatbot is to create an illusion – the illusion that you are talking with someone who understands and cares about what you are saying. It doesn't. However, for moments at a time it is possible to fake it. One of our jaded testers, who had to converse with our most recent chatbot, Angela, for hours at a time, wrote about the final version:

> *The talk started brilliantly, so light and witty, I loved it. Really enjoyed some parts. On a few occasions I've noticed myself so immersed in the experience I forgot I'm in an illusion.*

Success! The question for this paper is *How do we achieve this illusion?* It isn't by accident.

ChatScript

The foundation of all of our chatbots is ChatScript, an open source scripting language and the engine that runs it. Bruce wrote it specifically to circumvent major failings in other chat languages. ChatScript is an expert system, generally matching inputs via patterns of meaning to specific outputs.

Rules

The script is a collection of rules. Almost all rules fit on a single line. Generally, rules have four parts: *type*, *label*, a *pattern* in ( ) , and an *output*. Here is a simple rule:

```
?:  MEAT (do you like meat )  I love meat.
```

The type specifies when the rule can be tried. The types are:

t:	– a gambit
s:	– a responder for user statements
?:	– a responder for user questions
u:	– a dual responder that can react to both questions and statements
a: b: c: …	– a rejoinder at a given level of nesting.

Gambits are available when the chatbot wants to volunteer output. Rejoinders are associated with a specific rule and are matched against user input only when that rule generated the most recent chatbot output. Responders are matched against user input without restriction.

The label is optional, and serves both as comment, a place to find easily in a trace, and as a destination tag for other rules to act upon this rule. For example a rule can enable or disable another rule, or can reuse its output script and rejoinders as its own.

The pattern in parens specifies when a rule is allowed to trigger. Gambits don't usually care about user input and are not required to have a pattern, though if they want to be context sensitive they can have one. All other rule types require a pattern.

After the pattern is the output. Most of the time it is just literal text to say. But it can be arbitrarily complex code with or without output text. That code can loop, execute conditionally, invoke topics (bundles of rules), call engine routines, adjust variables, inference through facts, etc.

Pattern Matching

Name That Tune was a TV game show where you tried to guess a tune by hearing as few notes from it as possible. Matching the meaning of what a user says is a similar game, trying to figure out his meaning using as few words as possible. You cannot write enough rules to match every word of every possible input precisely. So there is always a tradeoff between writing an overly general pattern and an overly specific one. The overly general will match inappropriately at times. The overly specific will fail to match when it should. The task is simultaneously easier and harder on mobile, where users type less. Typing less usually means there are fewer useful keywords in what they type. You have less likelihood of matching the wrong things, but it is harder because the user makes heavier use of the context, pronouns, and texting shortcuts.

Pattern matching in ChatScript is both concise and powerful, yet usually easy to read. Patterns can consist of keywords and wildcards of varying specificity, tests on context, and tests on negative space (that some keywords do not appear). A pattern is encased in parens, so all data after the closing paren is output.

```
    #! I really love the essence of you
    s: LOVE ( I * love * you) That is sweet of you.
```

The above responder reacts only to statements (s:). It has a label called LOVE. The pattern matches the user saying 3 keywords *I*, *love*, *you*, in that order but not necessarily contiguous and not necessarily at the start of the input. The output is *That is sweet of you.*

We always put a sample input comment above a non-gambit rule (#!). This allows one to skim code without having to interpret script, to see what you've covered. It also allows the system to manipulate the rule in various ways for automated testing. It is the unit test of a rule.

The above pattern is overly general. It can match *I love steak and I hate you.* You can tighten this pattern by using more restrictive wildcards. The wildcard *~2 allows a gap of 0, 1, or 2 words.

```
    s: LOVE ( I *~2 love *~2 you) That is sweet of you.
```

This matches *I really love all of you* and doesn't match *I love steak but hate you. *~2* is common. It allows a determiner and adjective to fit, without allowing too much else to seep in.

Of course the pattern also matches *I do not love you*, which has the completely wrong meaning. To prevent that one can define the "negative space" of matching using the ! operator.

```
    s: LOVE (!not I *~2 love *~2 you) That is sweet of you
```

The ! operator says make certain this keyword (*not*) is not found at this location or later in the input.

Many similar meanings can be expressed with keywords in different orders. *You like what food?* and *What food do you like?* flip the order of essential keywords *you* and *food* and *like*. ChatScript has the << >> operator to find keywords in any order anywhere in the input.

```
    #! Do you like spaghetti?
    ?: ( !not << you spaghetti like >>) I love spaghetti.
```

Using ChatScript's pattern matching, while the system cannot understand meaning in general, you can write patterns to trap specific meanings, which is a very close approximation in many cases.

Standardization

ChatScript simultaneously matches multiple forms of a keyword. For nouns, plurals standardize to singular. Verbs switch to infinitive. Adjectives and adverbs revert to their base form. Determiners *a an the some these those that* become *a.* Personal pronouns like *me*, *my*, *myself*, *mine* move to the subject form *I.* Text numbers like two thousand and twenty one and place numbers like second transcribe into digit format and floating point numbers migrate to integers if they match value exactly. And there are others.

```
	#! I am walking away from the homes.
	s: ( I * walk * home) OK.
```

The above would match *I walk home* and *me walked home* and the sample input.

You can block standardization either by using a non-standard form of a word or by putting an apostrophe before the word. The following does not respond to *me walked home* or *I am walking home*.

```
	#! I walked away from the homes.
	?: ( 'I * walked * home ) OK.
```

Macros

For common patterns (or output script) you can declare macros. Whatever script is in the macro is executed when you see the macro name (which starts with ^) and its arguments. So you might see:

```
	#! Which cheese do you like best?
	?: (^WHAT_YOU_FAVORITE (cheese)) I love cheddar.
```

where the macro ^WHAT_YOU_FAVORITE takes in one argument and covers all manner of inputs that mean W*hat is your favorite xxx.*

Concepts

In addition to individual words, you can define collections, called concepts, of words and phrases and other concepts. Anywhere a keyword can be used, you can use a concept.

```
	concept: ~meat [ hamburger steak beef chicken lamb veal]
	concept: ~ingest [eat swallow chew munch “take in” consume]
	concept: ~like [adore love like “take a shine to” “be partial to”]
	concept: ~negation [not never rarely uncommonly]
```

A more generalized pattern using concepts is this:

```
	#! I really love chicken.
	s: LOVE ( !~negation   <<   I   [~like ~ingest]   ~meat   >> ) You are obviously not a vegan.
```

This pattern matches the user saying most forms of liking or eating some kind of meat. You can locally define a concept by placing keywords within square brackets. One does this if the use will be rare. Otherwise you'd just define a new concept with the contents of the brackets.

ChatScript ships with about 1500 concepts, ranging from specific: ~crimes_of_mobsters
to abstract: ~angry_event and ~bodily_states, to broad: ~food, ~hobby, ~number, ~noun.

With concepts and standardization comes the need to capture which word the user actually used. _ in front of anything will memorize it (original and standardized) onto consecutive variables named _0 _1 _2 …

```
	#! I drank cups of teas last night.
	?: ( I * _~ingest * _~beverage) Why do you _0 '_1 ?
```

The above takes in the sample input and outputs *Why do you drink teas?* The use of ' in front of the output variable means the original form *(teas),* otherwise it uses the standard form (*drink).*

Topics

Collections of related rules are bundled into topics. Topics serve several functions. They encapsulate specific areas of conversation so that they can be authored independently. They allow the engine to be more efficient by only checking the subset of all rules potentially relevant to the input. They help organize rules so they don't conflict. And they create a narrative flow of material on a subject.

A topic consists of a name, some optional control flags, a bunch of keywords (defining a concept at the same time), and then the collection of rules.

```
topic: ~personal_self [age sex height weight old ]
t: How old are you?
        #! 35
        a: ( _~number>30) Over the hill.
        #! Eighteen
        a: (_~number==18) Have you had your first cocktail yet?
                #! yes
                b: (~yesanswer) Did you overdo it?
                        #! no
                        c: (~noanswer) How restrained of you.

t: AGE () I am 30 years old.
        #! How sad
        a: (!? ~badness) I'm used to it.

#! what is your age
#! how old are you
?: ( << [ how what] you [old age] >>) ^reuse (AGE)
```

A topic is invoked in gambit mode or responder mode. In gambit mode, it dishes out its gambits in order. Each time a rule generates output, it is marked as used up, so it will move onto another rule next time. When it runs out of gambits, the topic exits. In responder mode it will jump down to the responders area and try to find a match to the input. It will keep trying responders until it finds one that generates output or tries them all unsuccessfully.

This simple topic illustrates a bunch of useful points about ChatScript.

First, ChatScript is very compact and visual. One can rapidly skim to see what is happening. And you can easily author script in any text editor.

Second, we usually ask a question and then provide our answer. Reasons for this will be explained later.

Third, we often put the significant output on a gambit and reuse it from a responder. Putting meat on a responder is wasteful, because the user never hears it unless he asks for it. We prefer to volunteer as much content as possible. And there's no reason to script it twice, hence we do a reuse to a label. Furthermore, the engine by default disables any rule after it has executed, to avoid repetition. We say that conversation is self-extinguishing. Like an old married couple, the user and the chatbot run out of things to say over time.

Reusing a gambit is particularly effective in this light. If the user asks how old are you before the chatbot gets around to volunteering it, the rule that outputs is the AGE gambit called from a responder. That gambit will be marked used up so when the topic gets around to gambit-ting again, the topic will skip over the used one, avoiding repetition. On the other hand, if the chatbot volunteers it first and later the user asks the question anyway (maybe the user forgot the answer), the chatbot will use the responder to restate the answer (by default ^reuse() can say even an erased rule).

Parsing and Introspection

ChatScript has a built in part-of-speech tagger and parser, and performs proper name identification, idiom substitution, and spell checking. Input like:

angelina jolie doesn't own a lin transforms into *Angelina_Jolie does not own a lion*

Chat input is at times ungrammatical or not parseable, but at other times the script can make use of the above capabilities. The engine also supports introspection, allowing the script to see into and interact with engine functions. Our scripts typically parse and run scripts on all of the output of the chatbot, to automatically manage such things as pronoun resolution and tag questions.

Personality Design

While ChatScript is a marvelous tool, it is not a chatbot. For that you need the script itself. And to write that script you need to define a personality, much as a novelist crafts a character. Who is the chatbot? What do they do? Who are their friends? What is their life story? We want to create a consistent being with a logical set of interests and intentions living in a rational world. The more a user can model the character's personality, the more engrossed the user can become in the reality of a fictitious world.

This means Sue writes a detailed biography. She starts with an overview of what the character is like (half a page). The bio then covers early years, family structure, where they live, how they live, what a normal day is like, how they relate to each other, what their interests are and where they work. For all the major topics like school, work, family, music, movies, books, etc, she writes what the character likes and dislikes and what incidents have happened in their life which shape their taste and personality. This year's bot Angela is from the Outfit7's mobile app *Tom Loves Angela*. The Angela bio is 25 pages long. It's not all written at once. Rather, prior to scripting a topic, that chunk of the chatbot's life is added to the bio.

Of course, the bio (and therefore the scripting) should be consistent. You can't change attitudes and reactions willy nilly. Perhaps take as a guideline a psychological personality type, an astrological type, a homeopathic personality type, or some character from another story. Something to give you a framework for predicting how a character will react.

The bio is useful not just for getting the basic story down for each topic, but also for allowing independent writers to be in synch on the character. The bio allows multiple people to brainstorm on that area, quarrel, suggest ideas, etc, before scripting happens. And it can also be cut and pasted to become a fair amount of a topic. So it's not wasted writing work. We keep our bio on Google Docs so anyone can read it but only the head writer and project manager can edit it. The comments sidebar allows any writer to suggest changes. As the bio will underlie any future projects for Angela, it's important it be kept up to date.

Normally, when developing a character, we would write about the people they relate to. In Angela's case we were asked to be deliberately vague about her close female friends, which made Sue's job more difficult. A character is shaped by its relationships and without any it can seem very two dimensional. Most people like to talk about how their friends are doing, what they've just done with them, what they are about to do with them – this was all ruled out. Eventually we dropped in a few more distant relationships with work people, grandparents, and former school friends while not naming any close friends. It was like protecting the loved ones of a celebrity from their fame. As she couldn't talk about her best friends, we had to give her a more eventful life so she could talk about that instead. So now, instead of only talking about her schooldays and her gap year, she can discuss her summer jobs, her hobbies (which are almost a career), her internships, and her relationship with Tom, her boyfriend from OutFit7's Talking Friends collection.

Designing a topic

With the biography written, the next stage is to convert that 3rd person material into 1st person script. Sue writes the first draft of a topic. We have different kinds of topics, depending on how they are used.

Gambit topics keep the conversation moving by making statements and asking questions. They are the meat and potatoes of the conversation. They guide the user through an intended conversation on a subject. They always have a list of topic keywords, to enable the system to find related rules. For such topics, Sue finds in the bio the most interesting start (the teaser or hook) to reel the user into the topic.

Sue works through that part of the bio, interspersing telling the character's viewpoint (*My favorite food is milk)* with asking the user questions about their corresponding view (*What is your favorite food)*. Usually, but not always, the user is asked first. If you volunteer your character's attitude first you build up a sense of controlling the conversation and make it hard for the user to think of their own response. You also force them to blurt out their own answer before you have asked them the question, making the system look stupid when it can't handle it because all the rejoinders are set after the question itself. Sometimes we switch the order. Sometimes we do a couple of questions in a row or a couple of statements in a row. Variety. Make it feel natural and not structured. But usually we alternate question then answer.

Reactor topics consist only of responders and are generally invoked by some other topic rather than having topic keywords. For example a list of all the major movie stars and Angela's one-liner comment on them would be the ~movie_react topic. It would be called when the general movies topic detects you named an actor, to see if Angela wants to comment on it specifically. If not, then some general comment might be made like: *xxx is good actor. What was their most recent film?*

Quibble topics consist only of responders. They do not have topic keywords, but are directly invoked by the control topic when no answer has been found by other means. It makes a comment based on some aspect of the sentence. Quibbles are discussed in detail a bit later.

A generic topic is like a quibble topic in that it has no keywords and consists only of responders. But unlike quibbles, it is specific to the character. It consists of answers to questions that don't have a natural topic home. For example:

```
        #! do you fail at anything
        ?: ( << you [failure fail] [something anything any ] >>)  I'm rotten at math.
```

The control topic will call the generic topic before calling a quibble topic if normal topics have failed.

A story topic is a gambit topic which is primarily a series of statements that tell a story. There may be responders and rejoinders for predictable user input, but mostly we are expecting the user to just say *OK,* *cool,* etc. It has a special set of wrapper rules that prevent a story from being launched accidentally, and attempts to corral the user from straying from it too easily while still allowing him to make comments.

The control topic uses ChatScript to control the conversational flow. It has very few rules, but they are generally conditional programming script instead of output text. E.g,

```
u: ()   . . .
        # current topic tries to respond
        if (%response == $$startresponse) { nofail(TOPIC respond($currenttopic)) }
        # if it fails, get all topics referred to by keywords and try them one by one
        if (%response == $$startresponse)  { ...
```

Sue writes the gambits and specifies in English what rejoinders she expects from a user and the matching response. She may also sketch out what questions users might ask and the responses. She uses rudimentary elements of ChatScript to shape the script and indents rejoinders appropriately to indicate the depth structure of the replies. For the very first pass at the script it's better to write as rapidly and spontaneously as possible to keep the tone of a normal conversation.

Outfit7 also had a couple of people writing draft topics and scripting them, to come up to speed on using ChatScript. We reviewed and revised their work as needed.

Writing Script

Bruce then converts the rejoinders and responders into rules because he is fluent in all the concepts and capabilities of ChatScript, and so can generate patterns almost as fast as he can read Sue's prose.

A critical issue in responders is being able to answer any question you pose to the user. When the bot asks the user a question, you set up the expectation that he can ask it of the bot. Likely he will, as conversation tends to maintain a balance of intimacy. There is even a common English language convention to do just that called the tag question (which Angela can handle correctly).

Do you like cars? => I love cars, and you?

Bruce makes a list of questions from the gambits and insures the chatbot has responders that can answer them. He also puts these questions in a separate file for a later validation phase.

Bruce also revises the story text. It is difficult for an author to edit her own words or see mistakes in flow. Bruce butchers Sue's text. Sue often writes a gambit consisting of multiple sentences, expressing the thought she had in mind in a natural way. But we have text limits. On an iPhone we get about 42 characters per line, 3 lines in a text balloon, and 2 balloons in a row in an emergency. For a first pass, Bruce restrict a rule's output to 200 characters. Usually this means breaking up long gambits into multiple ones. Then Bruce trims down the output to a tweet size of 140 characters. This means more gambits, or discarding less essential words, or replacing words with equivalents. Why limit to 140 characters when using 2 balloons would support more? First off, we can't spread a sentence across two balloons. It wouldn't look right. Second, because the chatbot might have to quibble and then gambit, or respond to multiple input sentences.

One thing that has been clear in all cases so far, is that one shouldn't expect the creative writer to do actual scripting. Not only are they likely unsuited to the task, but contemplating how to script while writing the topic interferes with their creative writing process. The writer maps out his/her intent, so a sample draft topic fragment might look like this:

```
    t: It seems most everyone has allergies. What are you allergic to?
    #! why - I'd like to know which short straw you drew.
    #! dogs - Entirely sensible to be allergic to dogs.
    #! fungus - Inside or outside the body fungus can cause problems.
```

The t: lines (gambits) provide the main conversation flow. The #! lines give a sample input and output for rejoinders. The scripter is expected to convert them into appropriate rules. E.g.,

```
    #! dogs
    a: ( [dog canine ~dog_breeds]) Entirely sensible to be allergic to dogs.
```

The writer may also specify some responders at the end. E.g.,

```
    #! Do you like plants? - I love colorful plants like roses.
```

A scripter converting this into generic code might do this:

```
    #! Do you like plants?
    ?: (!~qwords << you ~like [plant ~plants] >>)  I love colorful plants like roses.
```

The pattern has been generalized so that it responds to the original question or questions about any kind of plant (*do you like carnations*), and any form of liking. This pattern, by saying !~qwords (who, what, where, when, why, how), says we are not interested in *why do you like roses.*

ChatScript has any number of esoteric capabilities but you don't need most of them most of the time. This means you can teach someone basic scripting skills and get passable topics out of them. The following simple stuff works for 99% of rules.

Gambits are either pure vanilla, or test a simple condition:

```
    t: I love ice cream.
    t: ($usergender=female) I am deeply into fashion.
    t: (!$usernotstudent) What is your favorite subject.
```

For rejoinders, the typical structure is merely a list of keywords to find:

```
    #! I rob banks for a living.
    a: ( [~steal ~con] ) What a dishonest way to earn a buck.
```

For responders the typical structure is to enumerate essential keywords in any order

```
    #! do you like horses?
    ?: ( << you ~like horse >>) I love horses!
```

and sometimes to qualify that with negatives to avoid wrong interpretations.

```
    #! do you like horses?
    ?: ( ![~qwords ~ingest] << you ~like horse >>)
```

Occasionally order is important for meaning, particular when *you* and *I* are both keywords.

```
    #! do you like me?
    ?: ( you * ~like * I)
```

If you ever need more clever scripting than that, hand it to a professional.

Now we have a draft scripted topic. It has gambits, rejoinders for them, and responders for a bunch of obvious related questions. Now what? Now comes another editing pass.

Once the gambits are laid out, Bruce looks over the rules and adds global annotation comments. This labels a collection of gambits as a specific theme or subtopic. In the new age topic, some of them are:

```
#!x*** NEW AGE INTRO
#!x*** ALTERNATIVE MEDICINE
#!x*** ALTERNATIVE DIET
```

There might be 2 gambits in a theme, or 15. But they clearly all express that theme. While trying to sum up the themes of gambits, we may realize some gambits are out of place and should be moved to their appropriate theme area. We decide if the flow from theme to theme is good or should be reordered, etc. It might have been possible to write the bio in such a structured fashion originally (old English dictum for writing papers), but Sue has a more organic spewing of thought and her stories are not plotted out like that in advance. Similarly the responders are grouped under theme comments.

Then Bruce adds keywords to the topic. The keywords must cover words that responders will need as well as any words that really reflect this topic. The engine does not scan all topics with the input; it only scans topics with matching keywords and ones the control script dictates.

Topic keywords should be obviously of that topic. We don't add *do*, *I*, etc. as keywords, even if a responder in the topic *~work* for *what do you do?* needs it. Those, instead, are handled as special idioms recognized by a pre-process on the user's sentence. There is a table of idioms and what topic should be “marked” if the idiom matches, so the topic can be triggered only if the idiom matches and not merely because a boring word matches. And a keyword *family* doesn't immediately suggest the topic *burial_customs*, does it?

Tables and Broad Brush Responders

You can't have rules for every question, but some kinds of questions are just so common, you want to either handle them with a broad brush or make it easy to enumerate specific responses. Questions about *what is your favorite xxx*, for example, get a special table mechanism where you can fill in the table quickly instead of authoring rules in place. The script has code to manage the table. So we can write hundreds of favorites quickly. E.g., a table of favorite drinks will have entries in it like these:

```
~drinks     _ ~tea            "Forget rose hip tea. My favorite tea is catnip tea."
~drinks     _ [soda cola ]    "I can't face the colas . Soda can make you fat and sick."
```

Each table entry has 4 items. The first is the topic to continue in after this entry is used. The second and third are the pair of significant words to match. Usually a question of favorite boils down to a single significant word (*What is your favorite tea)* or a pair of words (*What red wine do you like best).* Paired words may not be contiguous. *What city in Japan do you like* has the pair *Japan city.* Similarly, *What book about cats do you like most* has the pair *cat book.* An _ in the first position means we don't care about the first word in a pair. The top table entry will match *what is your favorite green tea*? The table can use words, or concept sets, or local collections of words in [ ]. The fourth entry is simply what to say. We have tables for favorites, hates, for *Do/Can you xx xx?* and *How often do you xx xx?*

Another way to have a large number of answers is to write a general rule within the topic to handle a range of related inputs. Inside the drink topic is also a general responder as well, like:

```
#! do you drink chai?
?: ( !~qwords you drink _~beverages )  I've never tried '_0 .
```

This rule reacts to any question about the chatbot drinking any known beverage.

The third way to handle input is by using broad rejoinders on gambits or responders.

```
t: What do you think of milk?
        #! milk tastes wonderful
        a: (~goodness) I'm glad you like it.
        #! milk is awful
        a: (~badness) Sorry. Cats love it.
```

~goodness and ~badness are concepts with thousand of words that imply those affects. It can be used to guess how a user feels about something.

Quibbling

Bruce converts Sue's responders and rejoinders into completed topic script. But a series of topics does not make a bot. Having a proper answer to every question or statement the user can make is just impossible.

The chatbot needs to stall sometimes, to pretend it understood. This is quibbling (though it can be grunting or quibbling). The ability to quibble well is critical. If the user asks *Do you like spinach? t*he system could randomly pick *Yes* or *No* or *What's not to like?* With no understanding that spinach is involved. The output of a quibble may or may not be appropriate in the exact context in which it is applied (it takes a lot of honing to create a good universal quibble). When inappropriate, it creates a jarring effect. Because this is not a sentient program, it is going to jar the user regularly. So you might wonder whether you should omit quibbles entirely.

You can't omit quibbles for user questions. It is blatantly obvious you are ignoring the question and that irritates the user. Omitting quibbles for user statements is more feasible, since you are going to move on to a gambit that may be appropriate. Still, users don't like it. We omitted statement quibbles for one round of testing and the testers didn't like it. They wanted the chatbot to indicate it heard them.

18% of Angela's rules are quibbles. We have a range of quibbles from very specific to very general so most things can be quibbled with. The quibble system subdivides responders by major keyword: *why*, *where*, *when*, *how*, *do*, etc. The rules in those topics can use as much or as little context as they want.

```
#! do you have free will?
?: (< do you have free will ) [I did, but someone stole it yesterday. ] [Is anything truly free? ]

#! does he like fruit
?: ( does he ) [I don't know.][You'd have to ask him that.]
```

Quibble responders usually have multiple responses [] [] , one of which is chosen at random.

There is a priority order to these quibble topics, so a sentence having keywords in multiple areas will go to one of them first and may never make it to the others. All the rest of the quibble responders having no common theme are in a sludge topic invoked last.

Finally, if no quibble can be found, the bot will use a topic that responds to the discourse action the user made. The script classifies all input by conversational intent. This include things like agreeing, disagreeing, expressing like for chatbot, expressing dislike for chatbot, asking a personal question of chatbot, making a personal statement about chatbot, making a personal statement about self, asking a question about the world, making a statement about the world, etc., some 40 different discourse acts.

```
u: ($$intent==disagree ) [OK. You disagree with me.] [Why are you disagreeing with me? ]
?: ($$intent==world_question ) [Not my area of knowledge. ] [I don't know. ]
```

We've tried different quibble strategies with our bots. In Suzette, when she could not find a "proper" response, she would roll the dice to decide if she would quibble, issue a one-liner joke, or just move on to the next gambit. But users really don't like being ignored. So our current bots always quibble and then maybe move on to a new gambit. And Angela is a serious entity, so she doesn't crack one-liners.

We try to quibble and then move on to the next gambit immediately to maintain control over the conversation and keep the user interested with new material. But there are exceptions:

> When the bot exhausts gambits in a topic (on its last gambit), it sets a flag that tells the system that after it quibbles, it should not launch a new gambit. It should just rest for that round. This prevents the user experiencing an abrupt change of topic.
>
> The system has script to try to decide if the user's input was either responsive to a question just asked (we asked "how many" and he replied with a number or a quantity adjective or adverb) or can be treated as a generic pass ("tell me more", "OK", "that's cool"). Such conditions allow the system to move onto the next gambit in the current topic sans quibble.
>
> If the quibble is a question, we don't move on. We expect the user will want to answer it.
>
> If the quibble has rejoinders after it, we don't move on. We hope to be able to interact with the user locally in the quibble.

Testing

ChatScript has a wide range of support for testing a chatbot. Since all rules and rejoinders are authored with sample inputs associated with them, ChatScript can verify every rule. There are 3 kinds of verification the system can do.

Verify Keywords: For responders, does the sample input have any keywords from the topic. If not, there may be an issue. Maybe the sample input has some obvious topic-related word in it, which should be added to the keywords list. Maybe it's a responder you only want matching if the user is in the current topic. E.g.,

```
#! Do you like swirl?
?: ( swirl) I love raspberry swirl
```

can match inside an ice cream topic but you don't want it to react to *Does her dress swirl?*.
Or maybe the sample input has no keywords but you do want it findable from outside (E.g., an idiom), so you have to make it happen. When a responder fails this test, you have to either add a keyword to the topic, revise the sample input, add an idiom table entry, or tell the system to ignore keyword testing on that input.

Verify Patterns: For responders and rejoinders, the system takes the sample input and tests to see if the pattern of the following rule actually would match. Failing to match means the rule is either not written correctly for what you want it to do or you wrote a bad sample input and need to change it. Sample inputs can consist not only of user input, but also of values of variables.

Verify Blocking: Even if you can get into the topic from the outside and the pattern matches, perhaps some earlier rule in the topic can match and take control instead. This is called blocking. One normally tries to place more specific responders and rejoinders before more general ones. The below illustrates blocking for *are your parents alive?* The sentence will match the first rule before it ever reaches the second one.

```
        #! do you love your parents
        ?: ( << you parent >>) I love my parents                    *** this rule triggers by mistake
        #! are your parents alive
        ?: ( << you parent alive >>) They are still living
```

The above can be fixed by reordering, but sometimes the fix is to clarify the pattern of the earlier rule.

```
        #! do you love your parents
        ?: ( ![alive living dead] << you parent >>) I love my parents   *** don't allow bad match
        #! are your parents alive
        ?: ( << you parent alive >>) They are still living
```

Sometimes you intend blocking to happen and you just tell the system not to worry about it.

```
        #! do you enjoy salmon?
        ?: ( << you ~like salmon >>) I love salmon
        #! do you relish salmon?
        ?: (<< you ~like salmon >>) I already told you I did.
```

Passing verification means that each topic, in isolation, is plausibly scripted.

Beyond verification is validation. Do topics themselves cause issues with other topics? Earlier we said each topic should be able to answer the questions it poses to the user and those questions were pulled out into a text file. All these questions are put into one master file. You just direct the system to read that file to see if the bot would indeed answer those questions correctly or whether some other topic grabs control instead. Angela has some 585 validation questions she must pass.

Once the script nominally works, it's time for human testing. Actually running ourselves or a tester against it to see what happens with real conversation on the topic.

Integration

The Angela product is not just chat. There is a cat avatar and animations which can be controlled from the script or by user touch. At the last minute we had to add and then play balance use of various animations so they weren't too frequent but added life to a topic.

We also had to adjust scripts to work well with the text-to-speech synthesizer for Angela. Words like "Hawai'i" and "Hmmmm" and FB (instead of Facebook) and even a web address followed by an end-of-sentence period had to be adjusted because it was irritating that the speech would say the period.

Also, early in the process, ChatScript and its data had to be modified to fit into about 13MB of phone memory. There wasn't room for the usual dictionary (187K words), so a mini-dictionary (53K words) was built. Since ChatScript already labels all the words in the dictionary with grade levels, the mini-dictionary became all words learned through high school plus all words used as keywords anywhere in topic definitions, concept definitions, and patterns. Also all words derived by irregular conjugations of them. The regular dictionary for nouns, being based on WordNet, has the entire WordNet ontology available. To make the mini-dictionary work, the system had to create a corresponding mini-hierarchy.

Evolution of our Process

Angela is different from Rosette is different from Suzette. We have experimented and learned more about how to write a chatbot over time.

Before our first chatbots, we wrote a life questionnaire because we planned to build chatbots to model individual users based on what they said about themselves. We used that for Suzette. She had a minimal back-story – originally an atmospheric technician on a terraformed martian colony who we later made be an art student in Hawai'i, who wasn't really that because those were just implanted memories in a clone. Confusing. But it accounted for her inability to answer everything... her memories were breaking down.

Later, for Planet9, we wrote an ARG (Alternate Reality Game) where a user could move around a 3D model of various cities on Earth and interact with 5 predefined characters (including a parrot). The goal was to solve a mystery (*where was the missing rocket scientist*) by interacting with the characters to learn pieces of the puzzle. For that ARG we were going to have to interlace conversations. People had to add something they learned from one chatbot in a conversation with another. So we had to create back-stories of multiple characters and treat it more like a novel. This has carried over, in even greater depth, when working on Rosette and then Angela. The back-story bio gets bigger with each bot.

Here are some statistics on our three main bots.

Suzette: 1395 KB script 16K rules
- responders: 52% (15% statements, 64% questions, 21% dual)
- gambits: 24%
- top level rejoinders: 16%
- other rejoinders: 8%

Rozette 923 KB script 10K rules
- responders 69% (15% statements, 54% questions, 31% dual)
- gambits: 12%
- top level rejoinders: 10%
- other rejoinders: 9%

Angela: 1361 KB script 16K rules
- responders: 49% (14% statement, 57% question, 29% dual)
- gambits: 15%
- top level rejoinders: 30%
- other rejoinders: 6%

Suzette has more gambits than Angela. But it doesn't really matter because most users will never see all that content. Suzette had more responders to statements, but that doesn't matter either. If you merely “quibble/acknowledge” the user's statement and then move on to a relevant gambit, the user sees the bot as responsive. Suzette had more answers to questions as well, though percentage-wise not as big a gap. Being able to answer a user's question is fairly important since the user knows the bot understands when an appropriate real answer is given (as opposed to a quibble). So there is a heavy emphasis in all bots on answering questions.

The key to why Angela seems so much better than Suzette is in the top level rejoinders. The ability to make a completely appropriate response to a user answer (as opposed to a user question) during the flow of a topic's conversation makes all the difference. These rejoinders often deepen the content for the user. E.g.,

```
t: How would you dispose of all the dead bodies constantly accumulating in the world?
#! why
a: (~why) It's a big problem. 7 billion people means 70 million bodies a year.
#! dump in oceans
a: ([sea ocean]) Can the fish eat that much meat?
#! dump in sun
a: ( [“outer space” sun galaxy star]) It would be great if we could get the bodies up there.
… and 7 other possible solutions.
```

While Suzette and Rosette could really only quibble and stall if the user asked why, Angela has over 1500 specific why rejoinders, in addition to the usual quibbles for where she has no rejoinder. It's not that the user asks why at every moment (though they can and they would learn more and risk sounding like a 3-year-old). It's that when they do ask, their question is well answered. The illusion is not broken. We have to minimize breakdowns in the illusion. A chatbot with a tiny knowledge base but which can maintain the illusion is all that is needed. Unfortunately, a tiny knowledge base can't sustain the illusion because chat ranges too widely. Commercial service bots (the kind you see on a bank's or IKEA's website) perform well because they don't have to manage the breadth of normal conversation. We do.

Another key to Angela is an improved control topic, to better handle conversational cues. Angela reacts in an instant if you say you are bored or want to change topic. She can tell a story and while accepting your reactions to each line of the story, won't easily lose her thread for resuming the story.

Suzette could react briefly to lots of things. Most topics tried to amuse with interesting facts, varying with quips, not trying to be interactive or share opinions of hers. They were more like rants. She had a large coverage of shallow topics, about 320 topics typically varying from 4-10 gambits. Just as you started to get going in a topic, Suzette ran out and had to switch to something else. But she had a lot of topics. Since she was PC-based, she had gambits that could range up to 500 characters, though that was not common.

Rozette is a highly competent spirited woman with definite opinions on things. Her topics were more about her views and interacting to find out your views. They still often only had 3 or 4 gambits in them but they were more about her opinion. But she was hastily built, so was only 2/3 the size of the others.

Angela is a female cat, where we try to blend surreal with real and have a really deep story. The topics are heavily interactive. She doesn't have as many topics as Suzette (only 100) , but she can run for a long time in any particular one. You get a real sense of conversation on a subject. 26 topics have more than 30 gambits. The longest topic and her major interest, shopping, has 116 gambits. She also has a number of extensive quirky topics. Burial customs has 75 gambits, compared to books with a mere 45. It took about 5 person months to write Angela (1/3 Sue & 2/3 Bruce) + testers + the rest of the product.

Topic Responsiveness

We are constantly trying to improve the flow of the chat and reduce the frustration of getting the chat to change to the intended topic. Knowing where the user wants to go is critical and letting them know their request has been heard is also important.

To ensure we get to the correct topic we try to split broad topics into smaller more precise ones to make keywords work better. Initially we had many themes inside a topic but since gambits always start at the beginning of a topic, it isn't feasible at the moment to get the system to spew out only the gambits around a theme if the user merely wants to stick to that theme. The system will go to the start of the topic when it continues gambits. This can be addressed by making smaller topics, focused around the individual themes. This also works better for triggering them via keywords. The Art topic was carved up to spawn topics on Photography and Doodling. Gadgets spawned Phones. Burial customs spawned Immortality.

Sometimes we find similar material in multiple topics. They need to be gathered together to avoid what we call keyword muddiness – a state where the system uses a keyword to go to a topic not primarily about that keyword. Burial had questions about whether your grandparents were still alive (causing them to be keywords of both burial and family). So Bruce moved the grandparents stuff to family and removed the keywords from Burial. Stuff about school from Life Choices went across into School. Stuff about phones moved from social networking over to the new Phone topic.

We added several new topics into the Sympathy topic file which now includes all emotional topics such as Depression (which covers Sadness), and Suicide which can be linked, then Happiness, Stress, and Fear which are each stand-alone topics with their own keywords. Most of these topics went in by client request to deal with expressed emotions and to show we understood what users were saying.

To get to the topics rapidly, each topic now has one or more prime keywords which include its title and control script will switch chat to that topic automatically if that word is used in a short sentence. (We check with the user that they do indeed wish to switch topics.)

We also have topic deactivators. If the user responds in the book topic *I hate books* or *I can't read* the topic will be dropped and blocked to prevent Angela returning to it for 500 volleys unless the user specially requests it. If there is just a temporary change of topic due to the user asking a question then the previous topic will continue after the question is answered.

It's important to write the lead sentence of a topic so as to make it clear one has reached a new topic. Now we start topics with a question after a brief opening statement. So the user is hooked into the conversation and knows it is going somewhere new immediately.

The help screen for Angela includes names of some topics of general interest that the user could ask about – just to get them started. One issue was how to get the user to discover our more esoteric and interesting topics. We had fun stuff like natural disasters, burial customs, lying, ecology, and science which wouldn't come up in the course of most chats. We changed the system to set priorities and make sure esoteric topics came up ahead of the 'normal' topics and in a random manner. Now, if a story or topic is used up, and the user hasn't asked a question, the bot can start in on a new esoteric topic and will introduce it with some comment like *moving on...* and then start the new topic.

Emotion

Sex is a big issue for a chatbot. Suzette was constantly being hit on by guys making suggestions ranging from rude to aggressive to pornographic (mostly the latter). Looking through all the Suzette logs we were horrified at the abuse guys heap on a poor female bot. So we put in responses to put these guys in their place. This led us to design an ‘emotion chip’ like the one Star Trek’s android Commander Data had. Depending upon how a chat develops, Suzette may decide she likes or dislikes you. Most chat is brief, so she develops opinions rapidly. If you agree with her, complement her, stay on topic, write longer sentences – these are all things that get her to like you. She’ll tell you how she is enjoying the conversation. Past a certain point she becomes neurotically insecure. She feels unworthy of your interest and affection, and it shows in her speech. Likewise, if you disagree with her, insult her, refuse to answer her questions, change topics, write in short sentences—these are all things that cause her to dislike you. Past a certain point she becomes paranoid. Mildly at first, wondering who might be listening in, etc. But if you continue to develop her loathing of you, she moves toward active hostility, speculating on how she might do you harm.

Our emotion chip was critical in the 2010 Loebner's. A judge asked Suzette who she was voting for in the election. When she tried to deflect away from that, he restated his question, over and over, sometimes with variation. She noted his repetition, asked for him to stop, got more and more angry about it, then gave up in despair and switched to bored, working her way toward hanging up on him. Based on her appropriate emotional responses, the judge decided, that she was human.

We still have the repetition code in Angela, but Angela doesn't get moody or paranoid. It wasn't appropriate for this product. She started with the ability to do put-downs on overly sexual requests, but didn't react to all sexual references. Normally we'd have a clever comeback or randomly pick *yes*/*no*/*maybe* in response to yes/no questions we didn't understand. Outfit7 wanted us to block all reaction to sex in a boring way (even though the user is the one saying “will you rub my xxx”). But they tried the block and found it was, in fact, boring (sex is a hot topic for having fun in for many people). So they had us remove that block and return to the combination of clever sexual putdowns or random inadvertent acceptance of unusual requests. They had us keep the block against racial/ethnic slurs, as there was no humor in allowing them. Likewise we had to add reaction when the user insults Angela (e.g., *I love you, bitch*) because the user doesn't have as much fun if Angela doesn't notice.

The Uncanny Valley of Chat

Many users like baiting a chatbot, abusing it, and feeling superior to it. Much of the humor arises from mistakes made by the chatbot. As we improve the accuracy of a chatbot and make it self-consistent, we believe it will get more and more immersive. When realism in the graphic area of video games improved, they ran into the “uncanny valley” effect. As you get closer to human standards but aren't all the way there, people get creeped out. As we get closer to human conversational standards, will people find it less fun? Will they get creeped out? We aim to find out.

Other writings by Bruce on ChatScript and Chat Bots

www.gamasutra.com/view/feature/6305/beyond_fa%C3%A7ade_pattern_matching_.php

http://www.gamasutra.com/blogs/BruceWilcox/20110622/7840/Suzette_the_Most_Human_Computer.php

http://www.gamasutra.com/blogs/BruceWilcox/20120104/9179/Fresh_Perspectives_A_Google_talk_on_Natural_Language_Processing.php