ChatScript Finalizing a Bot Manual

Revision 10/16/2022 cs12.3

OK. You've written a bot. It sort of seems to work. Now, before releasing it, you should polish it. There are a bunch of tools to do this.

Verification (`:verify` and `:verifylist :verifyrun :verifymatch`)

There are two verification systems with slightly different purposes and abilities. They both depend on sample input comments.

When I write a topic, before every rejoinder and every responder, I put a sample input comment. This has #! as a prefix. E.g.,

```
topic: ~mytopic (keyword)

t: This is my topic.
    #! who cares
    a: (who) I care.

#! do you love me
?: (do you love) Yes.

#! I hate food
s: (I * ~hate * food) Too bad.
```

This serves two functions. First, it makes it easy to read a topic– you don't have to interpret pattern code to see what is intended. And you can use :abstract to give people a written abstract of your bot sans a bunch of actual code.

Second, it allows the system to verify your code in various ways. It is the "unit test" of a rule. If you've annotated your topics in this way, you can issue different :verify commands. You can run this regularly on your bot to prove you haven't broken things as you continue to modify it.

The basic `:verify` command can check your sample inputs in a number of ways (to be described) but has a limited ability to set up the conversation environment before the test. The more elaborate `:verifylist, :verifyrun, :verifymatch` trilogy has much more control.

Some bots can support multiple languages, changed on the fly. Or multiple personas that all share the same bot. Regressing these requires such control. One uses special VERIFY commands which can call functions or set variables during a test run. Here is a sample file which normally requires you be in english before you can use this topic (there are corresponding topics for other languages).

```
topic: ~SOCCER_en_US  system REPEAT (CHAT-TOPIC)
```

```
#! VERIFY ^setlanguagecontext(en_US)
u: () ^check_language(en_US)

#! What's your favorite soccer team?
?: (<<[best fave favorite] soccer [team club organization]>>) Madrid Real of course!

#! Do you like soccer?
?: (<<like soccer>>) Sure, what is your favorite club?

#! Who's your favorite soccer player?
?: (<<[best fave favorite]  soccer player>>) No one can bend it like Pele!

#! VERIFY ^setlanguagecontext(null)
```

When regression runs this topic, it will set the language to english, run the test, and then set the language back to what it was before.

These #! VERIFY comments can be stacked in succession and also accept:

```
#! VERIFY $var = value
#! VERIFY %time = value
#! VERIFY trace = all
#! VERIFY trace = none
#!  Do you like this?
```

To use this system, you start by doing `:verifylist` which will read your compiled bot and generate the file tmp/tmp.txt with top level rule commands to execute and expectations of where they end up.
It does not generate tests on rejoinders. Then if you say `:verifyrun` those commands are executed, generating a user log file (YOU MUST HAVE USER LOGGING ENABLED TO FILE). Then `:verifymatch` will generate a csv file whose columns are: Expected, Actual, Input, Persona, Response. Expected and Actual are rule labels. Input is from the verification sample input. If your bot as part of its oob output has a field of bot: and a botname (personality) then that is listed here as who said it. And finally the actual text of the response is shown. By default this file only contains failing tests. You can limit the csv to a specific language and/or bot and can request all tests be shown. The result of :verifyrun is the full data, so :verifymatch can be run multiple times to get different slices of the results.

```
:verifymatch language:en_us bot:roger all
```

There is a special verification input you can issue:

```
#! ~topicname $variable = value
```

When execution of a verify input is being done, when the topic with the name of the ~topicname sees this input, it will perform an assignment of the named variable with the named value. This is useful for enabling and disabling topics that might conflict. Eg.

```
topic: ~awe_en_US keep repeat ()
#! ~awe_event $event_context = awe_event
u: () ^check_event(awe_event)
```

Normally this topic checks to see if the event context has been set to the event for this topic. If not, the code in ^check_event executes a ^fail(TOPIC) and this topic will not react. But we WANT it to react during testing, so the comment will turn on this context to allow the topic to work (and other topics controlled by different events will not react during this topic's verify input.) At the end of the topic we have this:

```
#! ~awe_event $event_context = null
u: ()
```

Detailed :Verify

The :verifylist/:verifyrun/:verifymatch system has good control over the running environment but limits itself to telling you something doesn't match. You have to debug why yourself.

The simple `:verify` command can perform different test analyses.

Typically I start with proving the patterns work everywhere.

```
:verify pattern
```

The shows that all rule's patterns would actually match the sample input. At a minimum your pattern MUST do that.

Then I confirm keywords to access topics are OK.

```
:verify keyword
```

Then I verify rules don't block each other in a topic.

```
:verify blocking
```

Thereafter I can just do :verify to cover all of those.

Then I do more exotic checks:

```
:verify sampletopic
```

or

```
 :verify sample
```

will tell me if a sample would end up answered by some other topic. If I don't like the answer, then I need to fix that other topic's rule. Sampletopic just tells you if the rule was hijacked by a different topic. Sample tells you even if some other rule in the same topic hijacked you.

And I might do

```
:verify gambit
```

to see if the bot can answer questions it poses the user in gambits. You can also verify individual topics in various ways.

Here are more samples:

`:verify`

all topics, all ways but samples and gambits

`:verify ~topicname`

named topic, all ways but samples and gambits

`:verify ~to*`

all topics whose names start with ~to

`:verify keyword`

all topics doing keyword verification

`:verify ~topic pattern`

named topic, doing pattern verification

`:verify blocking`

all topics doing blocking verification

`:verify sample`

all topics doing sample verification

`:verify gambit`

all topics doing gambit verification

`:verify all`

all topics doing all verifications

Passing verification means that each topic is plausibly scripted.

To be clear how valuable `:verify` is, consider this example. We authored a demo bot for a company and they added a bunch of code themselves. It was a 5500 rule bot. When I got around to verifying it, the results were: 57 keyword complaints, 92 pattern complaints, 45 blocking complaints, and (after fixing all those other complaints) 250 sampletopic complaints.

The `:verify` command takes the topic name first (and optional) and then a series of keywords about what to do.

`:verify keyword`

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 entry to send you to this topic, or tell the system to ignore keyword testing on that input.

You can suppress keyword testing by augmenting the comment on the sample input with `!K` or marking the topic as a whole `TOPIC_NOKEYWORDS`

`#!!K` this is input that does not get keyword tested.

`:verify pattern`

For responders and rejoinders, the system takes the sample input and tests to see if the pattern of the following rule actually would match the given input. 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. This test SHOULD NOT fail when you are done. Either your sample is bad, your pattern is bad, or ChatScript is bad.

Rules that need variables set certain ways can do variable assigns (`$` or `_` or) at the end of the comment. You can also have more than one verification input line before a rule.

```
#! I am male $gender = male
#! I hate males $gender = male
s: ($gender=male I * male) You are not my type
```

For a pattern that starts with `@_3+`, you can set the position by an assignment naming the word to start at (0 ... number of words in sentence).

```
#! I am the leader @_3+=0
```

If you want to suppress testing, add `!P` to the comment.

```
#!!P This doesn't get pattern testing.
```

If the pattern absolutely SHOULD fail the test, use F. As in `#!F` or `#!!FB`. If you want to suppress pattern and keyword testing, just use `K` and `P` in either order:

```
#!!KP this gets neither testing.
```

You can also test that the input does not match the pattern by using `#!!R` instead of `#!`, though unless you were writing engine diagnostic tests this would be worthless to you.

`: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

#! 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 using `!B`. Or you can have the entire topic ignored by the topic control `TOPIC NOBLOCKING`.

```
#! do you enjoy salmon?
?: ( << you ~like salmon >>) I love salmon

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

The blocking test presumes you are within the topic and says nothing about whether the rule could be reached if you were outside the topic. That's the job of the keyword test. And it only looks at your sample input. Interpreting your pattern can be way too difficult.

If `:trace` has been set non-zero, then tracing will be turned off during verification, but any rules that fail pattern verification will be immediately be rerun with tracing on so you can see why it failed.

```
:verify gambit
```

One principle we follow in designing our bots is that if a user is asked a question in a gambit, the bot had better be able to answer that same question if asked of it.

The gambit verification will read all your gambits and if it asks a question, will ask that question of your bot. To pass, the answer must come from that topic or leave the bot with that topic as the current topic.

You can stop a topic doing this by adding the flag `TOPIC_NOGAMBITS`. There is no way to suppress an individual gambit, though clearly there are some questions that are almost rhetorical and won't require your bot answer them. E.g., *Did you know that hawks fly faster than a model airplane?*

```
:verify sampletopic & :verify sample
```

Once you've cleaned up all the other verifications, you get to sample verification. It takes sample inputs of your responders and sees if the chatbot would end up at the corresponding rule if the user issued it from scratch. This means it would have to find the right topic and find the right rule. If it finds the right topic, it will usually pass if you have managed most of the blocking issues.

I start with `:verify sampletopic`, which only shows inputs that fail to reach the topic they came from. This is generally more serious. It may be perfectly acceptable that the input gets trapped in another topic. This sample result is merely advisory. But maybe the rule that trapped it should be moved into the tested topic. Or maybe its just fine. But if the output you got doesn't work for the input, you can go to the rule for the output you got and alter the pattern in some way that excludes it reacting inappropriately to the input.

Once that is cleaned up, `:verify sample` will include rules that fail to get back to the correct rule of the topic. Usually, if you've fixed blocking, this won't be a problem.

If you stick a rule into Harry's `~introductions` topic like:

```
#! what are you
u: (what are you) I am a robot
```

Then `:verify sample` will complain about the sample. It is saying that if you are not ALREADY in the topic `~introductions`, then the input you give will get answered by some other topic (quibble) and not by this topic. The reason this would be true is that there are no keywords in the `~introduction` topic that could allow it to be found if you are not already in this topic.

Normally, a question like *what are you* is something I would put in the `~keywordless` topic, because there is no natural keyword in the question and you could be asked that question at any time from any topic. If, however, you actually want this rule in `~introductions`, you can tell CS NOT to do the sample input test on it via

```
#!!S What are you
```

Changing tokenization

`:verify` normally works on the current `$cs_token` value. But in my bots, I might process inputs using two different values, one initially and one later. You can optionally specify what tokenization to be using by naming a user variable first.

```
:verify $mytokencontrol ~mytopic pattern
```

Verify variables are assigned and used

':verify variablereference' will list all user variable assignments and uses in a bot into TMP/reference.txt . If you sort that file, you may detect places where you misspelled a variable, assigned but didnt use, or used but didnt assign. Normally it only shows mismatches but ":variablereference full" will list all variables.

It may miss some assignments when those are done indirectly, e.g.,`$data.$_category.brand = Amana` means it may not list an assignment for `$data.appliance.brand`.

Spelling

To ensure the outputs don't have spelling errors, I do a `:build` like this: `:build Harry` outputspell and then fix any spelling complaints that should be fixed.

Sizing

We generally adhere to a tweet limit (140 characters) and run `:abstract 140` to warn us of lines that are longer.

':listvariables'

Because variables in CS have only 1 data type (text string), CS does not have any predeclarations of variables. You just use them on the fly. Of course if you make a typo, you use that on the fly as well and it may not be what you are expecting. Hence :listvariables.

:listvariables analyzes your compiled code in the TOPIC folder and writes a file listing each instance of a variable you have. The file is TMP/variables.txt. By default it lists all sorts of variables but you can limit it by naming the kinds you want:

```
all - all variables
local - $_ variables
global - $ and $$ variables
permanent - $ variables
transient - $$ variables
```

Each line of the file names the variable and puts a + after it if it was an assignment. If you then sort this file (e.g. DOS sort command 'sort TMP/variables.txt > x.txt) they are made suitable to then from CS run :mergelines x.txt which merges all lines having the same starting word so there is just 1 example of each line with a prefix count of how many instances of such lines were seen. If you then sort that result (eg sort TMP/tmp.txt y.txt) you get a file that gives you how many times a variable was used. Things used only once are highly suspect

(set or used without the corresponding other side). Helps you see instances where maybe you typed the misspelled your variable.

`:mergelines file {all}`

The function outputs to tmp/tmp.txt. It reads in lines and for adjacent exact matches, it outputs a single line with a count in front. This you can then use extern sort routine (eg sort tmp.txt >x.txt) to order the file by frequency of occurence. Normally :mergelines compares just the first word of the line, which is good if lines start with their rule name. If you add the optional all argument, then it is good if either user input or bot output lead the line.

I use this to find unique user inputs (low count) and see if they are well handled.

`:topicinfo ~topic how`

This displays all sorts of information about a topic including its keywords, how they overlap with other topics, what rules exist and whether they are erased or not. You either name the topic or you can just use `~`, which means the current rejoinder topic (if there is one). You can also wildcard the name like `~co*` to see all topics that start with `~co`.

If 'how' is omitted, you get everything. You can restrict things with a collection of how keywords. These include:

how	meaning
`keys`	to display the keywords
`overlap`	to display the overlap with other topics'
`keywordsgambitsrejoindersrejoinders1`	to some of those
`usedavailable`	to see only those rules meeting that criteria.

Of particular importance in finalization is the key overlap map. Keyword overlap is particularly interesting. As you assign keywords to topics, at times you will probably get excessive. Some topics will share keywords with other topics. For some things, this is reasonable. "quark" is a fine keyword for a topic on cheese and one on astronomy. But odds are "family" is not a great keyword for a topic on money.

Often an extraneous keyword won't really matter, but if the system is looking for a topic to gambit based on "family", you really don't want it distracted by a faulty reference to money. That is, you want to know what keywords are shared across topics and then you can decide if that's appropriate. Sometimes you are told a word but don't see it in the topic keywords. Use `:up` on that word to see how it intersects.

I use `:topicinfo` keys to generate a map across all topics. I then read the first column list of keywords for each topic to see if they obviously should be in that

topic or if they don't really strongly imply that topic. If not, I try to remove them from it.

Sometimes when I look at the list of topics that a key overlaps with, I find multiple topics with similar ideas. Like a topc called `~cars` and one called `~automobiles`. Often they are candidates for merging the topics.

:describe and :list

If you have used :describe to document all of your permanent variables (see advanced CS, `:build`), then `:list` can be used to show undocumented and potentially erroneously spelled permanent variables

```
:list $ ^ ~ _ @
```

will list the documented kinds of items (you name which ones you want, could be just $) and for variables it will list be documented and undocumented.

:abstract

There are several useful :abstract calls to do during finalization.

```
:abstract 100
```

If you want to adjust output of yours that would be too long for something like a phone screen, you can ask :abstract to show you all rules whose output would likely exceed some limit (here 100). Without a topic name it does the entire system. With a topic name e.g.

```
:abstract 100 ~topicname
```

it is restricted to that topic. You can also use a wildcard like `~top*` to do all topics with `~top` at the start of the name. You can also provide a list of topics, like

```
:abstract ~topic1 ~topic2
```

```
:abstract censor ~mywords
```

will note all output which contains any words in `~mywords`. Of course regular uses may also appear. The censor command looks for any words referred to by the concept given.

```
:abstract spell
```

will examine the outputs of all topics (or topic given) to find words whose spelling might be faulty. It's not a guarantee it is, but it can warn you about potential mistakes. And `:abstract` is handy just to print out a human-readable copy of your bot, without all the ChatScript scripting mess. As part of that, you can add header comments that will appear in the abstract. These are done as:

```
#!x whatever
```

I often subset gambits and responders in a topic by section, e.g.,

```
#!x*** FAMILY COMPOSITION
...
#!x*** FAMILY RELATIONSHIPS
```

:abstract story

will display just topics and their gambits (+ rejoinders of them).

:abstract responder

will display topics and their responders (+ rejoinders of them).

:permanentvariables

This gives you a list of all permanent variables found in your source. Not only is documenting them a good idea, but you might want to confirm they are all legit (correctly spelled) and reinitialized as appropriate.

Regression

Having built a functioning chatbot with lots of topics, I like to ensure I don't damage old material in the future, so I create a regression test.

There are two ways to make a regression test (and you can have multple tests). One is to create a source file with a bunch of questions. To be safe from most randomization effects, you should designate the user and :reset the bot. After doing a bunch of stuff you can do :reset again, so additional inputs are immune from effects of the prior stuff.

You can see an example of my std Rose regression file in `REGRESS/chatbotquestions/loebner.txt`

It starts out like this:

```
:user test
:reset
# 2001 Chatterbox challenge
How is it going?
How old are you?
Are you male or female?
What is your favorite color?
Are you a bot?
What is the date?
...
:reset
```

and then has more questions and more :resets. Eventually it has a `:quit` before a bunch of stuff I'm not wanting tested at present.

The alternative way is to start a fresh conversation as a new user and assuming it makes no mistakes then stop. You need to be a completely new user because CS will read your log file and things like `:reset` don't clear it. You want to have only the conversation you just did in that log. Now you can make a regression file by typing `:regress init test` (or whatever user name you used from either mechanism). CS reads your log file and converts it into `TMP/regress.txt` or you can do

```
:regress init user outputname
```

to name where it goes.

The user can be a full file name or just the name of a user whose log is in the USERS directory. The output name should be whereever you want to put the resulting regression file. If omitted it defaults to `TMP/regress.txt`. Thereafter, all you have to do (not from a server version) is

```
:regress outputname
```

And the system will retest your conversation. I usually tranfer the file over to a regression directory and rename it in there.

Regression will note any differences. Inconsequential differences it will merely tally at the end, more interesting differences which may or may not be correct show the alternative outputs and info, and differences CS thinks are major are flagged. If you use

```
:regress terse outputname
```

then the system will only note what it thinks are fatal differences. If there are differences, maybe they are correct (you edited your code). But CS can detect where you merely edited the output, or a pattern, or a label, or added or subtracted rules nearby. So it is likely to tell you what you need to know.

The system will ask if you want to update your regression file, and if you answer "yes", it will rewrite the regression file with the updated test results. Do this if it detects minor differences so it can help stay on track in the future. Do this if the major differences are in fact OK. Otherwise decline and go fix your code somewhere.

```
:regress batch outputname
```

Useful for batch applications, this form does not prompt you if stuff is found changed. Instead it causes ChatScript to exit (0) if regress passes, or exit(1) if it doesnt. ## Mobile size issues

Mobile apps embedding CS probably do not want the entire dictionary. CS ships with a much smaller dictionary supporting basic english (through grade 6) in

the folder DICT/BASIC. For a mobile app, all you need is a DICT/ENGLISH folder with the dict.bin and fact.bin from the BASIC folder.

CS will add things it finds in your TOPIC folder into the dictionary when it loads, so you can supplement words (spelling and parts of speech from those). They will not, however, provide correct pos tagging on conjugations of words not in the basic dictionary. And spelling correction may or may not work properly for words not in the dictionary. And the wordnet ontology will not propogate data through words not in the dictionary into higher level mappings. Eg Doberman may not know it is a more refined word of dog (though the concept sets will work). If you find you need a better dictionary, for a fee I can give you a basic dictionary where all the words of your script patterns and concepts are also correctly mapped.

You can also reduce code size by declaring a bunch of DISCARDxxx defines (see start of SRC/common.h) to remove chunks of CS you don't need, like the ScriptCompiler or testing abilities.