TestNLP: An Worked Example (Using E-mail A Friend)

Jeff Nyman

[TestNLP was originally created at Classified Ventures. I do retain full rights to TestNLP and its derivative work, "Natural Testing." Any examples here that are specific to Classified Ventures are general enough as to not breach any non-disclosure agreements.]

TestNLP is based on a field of study within the testing discipline called model-based testing. TestNLP is also based on the idea of rules-based heuristic engine. The goal of TestNLP is to see how much a natural language approach could work in bridging the specifications, the test conditions and the automated tests.

A central tenet of this approach is that testing is a form of narration; a system for writing test conditions and deriving those conditions from requirements can be judged by the range of meaning it narrates. This ties in with the idea of narrating complex circumstances with clarity. That is largely what writing business and/or use cases is about and also what deriving test scenarios from them is about. The easy part is taking specification words and translating those into different words. The harder part is taking those words and translating them into combinations of test conditions and ultimately into automated code. And yet many features of natural language are readily imitated in conventional code: a verb juxtaposing nouns is a procedure call with its arguments, an adjective is a function which returns true or false, a proper noun is a constant (or an object), a common noun is a data type (or a class), prepositional phrases could be regarded as tests of standard data structures (such as "if...then" constructs, for example).

Testing is based on a dialogue between tester and developer, with both directions of communication prompted by textual possibilities supplied by a document (business specification or technical specification). In general, the idea then has three different expressions:

> Business specification [business/analyst]
> Technical specification [analyst/developer]
> Manual test scenarios/conditions [tester]

If you add automation into that, you have yet one more expression.

> Automated test scenarios/conditions [tester/developer]

Words are just words, however; it is repetitious and artificial to have to write them differently all three (or four) ways. The idea is that natural language collapses some of the separation. *That* is what TestNLP is predicated upon. TestNLP tries to move

everything from an object-oriented to a rule-oriented model in the same way that you might consider the distinction between an object-oriented language and a relation-oriented language. TestNLP has three guidelines:

(1) Any domain knowledgeable reader should easily be able to guess what a sentence says and that guess should stand a very good chance of being correct.

(2) The language should be somewhat constrained but not so much that it degenerates into pseudo-code on the one hand or unintelligibility on the other.

(3) Contextual knowledge is best supplied by the writer of a given deliverable rather than being "built in."

What (1) and (2) speak to is that the language is predicated upon natural language that people use all the time, but in a more constrained format. Writing legal documents, for example, entails the use of natural language but no one would argue that the language used is purposely constrained to be very concise. In a similar fashion, that is what specifications are supposed to be: the unambiguous conveying of ideas for implementation. The "unambiguous" qualifier applies to (3) because specifications should not be written for only those who already understand: necessary context must be provided if that context will affect implementation or understanding.

By necessity, I am going to offer a simple example from the E-mail a Friend functionality of the Search Consumer Site (SCS). Here I am just concentrating on the specification wherein it describes the pop-up window and interactions on that window.

[**Reference:** http://hs/techspec/scs_v6.7/func_specs/ts_emailafriend_v6.7.doc, *Section 3.2 - Content*]

What I am going to do is show how I rewrote that part of the specification in such a fashion that it allowed me to automatically derive test cases of a manual sort and, since it was applicable, of an automated sort.

The plan of this document is as follows:

Section 1.1: Source Text. This will give a full "source text" example for use in TestNLP.

Section 1.2: Source Text – With Comments. This will give the same full "source text" from section 1.1 but comments will be included within the text itself to call attention to some specifics.

Section 1.3: Generated Elements. This section will show how the source text is converted by TestNLP into manual test cases and into automated code.

Section 1.4: Other Uses. This section will show a few other uses that I have put my TestNLP to.

1.1 - Source Text

Here is an example "source text" for input to the TestNLP engine. I show this so that it is possible to get a feel for what such text looks like as a whole. What you have to imagine, in this case, is that the text given below is a way to write the specification itself, in terms of business requirements or technical requirements.

3.0 E-mail a Friend Pop-Up
3.2 Content

Understand "E-mail A Friend Pop-Up Window" or "E-mail A Friend Pop-Up" as "EAF Popup"

3.2.1 Page Header

Spec: EAF Popup must display header text from site definition. Check site definition: SITE_ATTR_TYPES.ATTR_TYPE_NAME.

Rule: If site definition has element FRIENDDSPLTXT (called Display text for E-mail a Friend), the text to use is text value of "[Display text for E-mail a Friend]".

Rule: If site definition has no value or site definition has value that is not FRIENDDSPLTXT, the text to use is nothing.

3.2.2 Instructional Text

Spec: EAF Popup must display instructional text (called text).

Rule: The text is "Share details about this property with one or more people by entering their e-mail address(es) and any message below."

3.2.3 Recipient E-mail Address

Spec: Recipient E-mail Address must exist on EAF pop-up as a text field.

Rule: Label for text field must be "Recipient E-mail Address(es):".

Rule: E-mail Address field must allow multiple e-mail addresses.

Rule: Multiple e-mail addresses are separated by a comma.

Rule: Multiple e-mail addresses are separated by a semicolon.

Rule: E-mail Address field must allow text entry.

Rule: Text entry must be limited to 4000 characters.

Rule: Recipient E-mail Address is required.

Rule: Recipient E-mail Address must check validation.

3.2.4 Sender's Name

Spec: Sender Name field must exist on EAF pop-up as a text field.

Rule: Label for text field must be "Your Name:".

Rule: Sender Name field must allow text entry.

Rule: Text entry must be limited to 255 characters.

3.2.5 Sender's E-mail Address

Spec: Sender E-mail Address field must exist on EAF pop-up as a text field.

Rule: Sender E-mail Address must check validation.

Rule: Label for text field must be "Your E-mail Address:".

Rule: E-mail Address must allow one e-mail address.

Rule: E-mail address is required.

Rule: Text entry is limited to 255 characters.

Rule: E-mail Address field must allow text entry.

Rule: E-mail Address must not allow multiple e-mail addresses.

3.2.9 Send

Spec: Send must exist on EAF pop-up as a button.

Rule: Button text must be "Send".

Procedure: Clicking button performs validation.

Check validation:

begin;

   Recipient E-mail Address must be populated.
   Sender E-mail Address must be populated.
   Recipient E-mail Address - apply valid formatting for e-mail address;

Sender E-mail Address-apply valid formatting for e-mail address;

otherwise;

Error: "Please enter a valid recipient e-mail address to process your request."
Error: "Please enter your e-mail address to process your request."
Error: "One or more e-mail addresses are invalid. E-mail addresses should have the format user@domain.ext [,user@domain.ext,…] to process your request."

end.

repeat with EAF Popup
Check "[required field arrow]" (called icon) for any field that is required: icon is left of label of field.
end repeat;

1.2 - Source Text (With Comments)

Here is the source text from section 1.1 again but commented. In TestNLP, comments are interspersed as text in square bracket, [ ], characters. If text is used in square brackets in a string, as in "[text]" those are not comments, but text substitution indicators. Note that any given comment is generally going to be talking about the line(s) that follow, not those that precede.

[Anything with a non-spelled out number in front of it will be treated as a section. If you spelled out the number, TestNLP tries to figure out what you meant and will not treat that as a section. For example, say you had an assertion like this:

    One e-mail is allowed.

TestNLP would not treat that as section one (1) but rather as an assertion indicating something about an object known as an e-mail. So the two elements below designate sections. TestNLP will attempt to determine if numbering appears to be out of order or contains gaps. For example, if there was not a 3.1, TestNLP would warn about that.]

```
3.0 E-mail a Friend Pop-Up
3.2 Content
```

[The following "understand" assertion is used to allow "EAF Popup" to be used as a synonym or alias for longer terms. Technically this is not necessary as you could just use the longer terms, but it can make your life easier when typing out a specification. This basically is similar to how participants in a meeting might agree to a shorthand notion for referring to something.]

```
Understand "E-mail A Friend Pop-Up Window" or "E-mail A Friend
Pop-Up" as "EAF Popup"
```

```
3.2.1 Page Header
```

[In the Spec line below the "from" may seem a bit odd but I have to use it because it indicates that the EAF Popup object must do something. What must it do? It must display header text. From where? From the site definition.

Note that the "Check site definition" assertion sets up a new object called site definition. The assertion, which is the Check command begins when the colon (:) is reached and ends when a period is encountered.]

```
Spec: EAF Popup must display header text from site definition.
Check site definition: SITE_ATTR_TYPES.ATTR_TYPE_NAME.
```

[The first rule below may seem kind of cumbersome but that is because I wrote it to match what our current specification looks like. Our specification has an item (1.1.2) in section 1.1 called Display text for E-mail a Friend. You are supposed to know to look

back that that whenever that text is referenced. I could actually rewrite my Rule as such:

Rule: If site definition has element FRIENDDSPLTXT, the text to use is text value of FRIENDDSPLTXT.]

Rule: If site definition has element FRIENDDSPLTXT (called Display text for E-mail a Friend), the text to use is text value of "[Display text for E-mail a Friend]".

[Note that in the second Rule:, you can use null or "" instead of nothing.

Also, notice how I do not split off the site definition elements from the content elements as we do in our current specifications. Everything is kept logically together. However, I do have provisions in place in TestNLP to handle different sections. So you could have something like this at the top of the text:

Definition: SITE_ATTR_TYPES.ATTR_TYPE_NAME (called HeaderTextSD) applies to display header text; acceptable value is FRIENDDSPLTXT (called DisplayTextSD).

Here the "called" allows you to refer to SITE_ATTR_TYPES.ATTR_TYPE_NAME easier. Then you could rewrite, for example, some elements of the above spec assertion:

Check site definition: HeaderTextSD
Rule: If site definition has value DisplayTextSD, the text to use is "[Display text for E-mail a Friend]".]

Rule: If site definition has no value or site definition has value that is not FRIENDDSPLTXT, the text to use is nothing.

3.2.2 Instructional Text

[Note that the Spec and Rule below could be written as a specific Check assertion that gives guidance of what should happen if the rule is broken. However, since the rule only says that certain text should appear, TestNLP assumes that if that specific text does not appear, a problem should be reported.

Also note the (called text) is not needed. If you did not do that, then the rule could start off like this:

> Rule: The instructional text is... or could even be kept as it is.

The reason "text" by itself would work is because TestNLP takes everything in context. Since the Spec: mentions "instructional text", a rule applied to just "text" will be assumed to be referring to the same thing, unless some other rules give TestNLP a reason to question that assumption. For example, say you had a rule like this between the current Spec and Rule:

> Rule: Error text should appear.

In this case, you would have to specify that you mean instructional text because text alone would not be enough to help TestNLP differentiate between "Error text" and "Instructional text".]

Spec: EAF Popup must display instructional text (called text).

Rule: The text is "Share details about this property with one or more people by entering their e-mail address(es) and any message below."

3.2.3 Recipient E-mail Address

[Notice below that I reference not just that something must exist but also how it should exist: meaning, what form it should take. I could have also worded the Spec line as such:

> Spec: A text field must exist on EAF pop-up called Recipient E-mail Address.

TestNLP reads the lines in the same fashion. Note that Recipient E-mail Address becomes an object in TestNLP in both cases. I could have also done this:

> Spec: Recipient E-mail Address must exist on EAF pop-up.
> Rule: Recipient E-mail Address is a text field.

Notice that I do not have to say "recipient" in all the rules because TestNLP assumes that is what is meant since the rules will apply to the last Spec assertion. However, you could use the full name if you wanted. You could actually even just use Address since that phrase only is used in the context of the field itself. However, you should not use just E-mail as a reference because the rules do utilize that term in two ways: one as the field itself and another as what gets entered into the fields. See the second Rule, for example.]

Spec: Recipient E-mail Address must exist on EAF pop-up as a text field.

Rule: Label for text field must be "Recipient E-mail Address(es):".

[Looking at the second Rule, how does TestNLP know what an e-mail address is? And how does it distinguish that general concept from the specific concept of E-mail Address, which is the field being discussed?

TestNLP has a global Definition assertion that looks like this:

> Definition: An e-mail address is a text value that follows the pattern "text@text.text".

That is how TestNLP "knows" what an e-mail address is. As to how it distinguishes between the field and the type of data, this is done by context. Notice now the second Rule says "E-mail Address field"? The "field" helps TestNLP disambiguate.

With this being said, it is recommended that "Recipient E-mail Address" be used, in full, when referring to the field and "e-mail address", or the plural form, is used when referring to actual e-mail data. This will help TestNLP disambiguate much easier.

Speaking of the definition idea, notice the first rule? TestNLP also has a global definition like this:

> Definition: A label is text to the left of a field.

These definitions can be overridden in specific cases, if you needed to.]

Rule: E-mail Address field must allow multiple e-mail addresses.

Rule: Multiple e-mail addresses are separated by a comma.

Rule: Multiple e-mail addresses are separated by a semicolon.

Rule: E-mail Address field must allow text entry.

Rule: Text entry must be limited to 4000 characters.

Rule: Recipient E-mail Address is required.

Rule: Recipient E-mail Address must check validation.

3.2.4 Sender's Name

Spec: Sender Name field must exist on EAF pop-up as a text field.

Rule: Label for text field must be "Your Name:".

Rule: Sender Name field must allow text entry.

Rule: Text entry must be limited to 255 characters.

[What follows below is pretty much the same thing as for recipient e-mail address. Note that the order of the Rules does not actually matter all that much. You will see, for example, that here the rules are in a different order and not necessarily logically grouped. At least from TestNLP's point of view, that is not a problem because it reads the whole set as one and holds off making decisions until the entire rule set for a given Spec identifier has been parsed.]

3.2.5 Sender's E-mail Address

Spec: Sender E-mail Address field must exist on EAF pop-up as a text field.

Rule: **Sender E-mail Address must check validation.**

Rule: **Label for text field must be "Your E-mail Address:".**

Rule: **E-mail Address must allow one e-mail address.**

Rule: **E-mail address is required.**

Rule: **Text entry is limited to 255 characters.**

Rule: **E-mail Address field must allow text entry.**

Rule: **E-mail Address must not allow multiple e-mail addresses.**

[The Send portion is a bit different than what has gone before...]

3.2.9 Send

Spec: **Send must exist on EAF pop-up as a button.**

Rule: **Button text must be "Send".**

[The next line is a Procedure assertion and this indicates to TestNLP that what follows is not something that describes how the application looks, but rather than an actual action must be performed. Note that "validation" is the same identifier I used in the some of the above rules, such as when I said things like:

> Sender E-mail Address must check validation.

Here since I only have one "validation" TestNLP actually manages to figure out what is meant here. However, this is not good practice. I showed everything this way just to show it can be done. In reality you would want to probably identify the validations, as such:

> Sender E-mail Address must check Send validation.

You can also give validation steps outside of any Spec areas. In fact, that is sort of what the code here really is: the validation simply looks like it is part of the Spec section because it follows from it.]

Procedure: **Clicking button performs validation.**

[This is the part that is the most "code-looking" right now in terms of forcing the writer of the specification to think outside the strict dictates of natural language. Here each element in the begin section that ends with a period should have a corresponding entry in the otherwise section. Note, however, that the last two lines in the begin section are separated by a semicolon. This means they are considered to generate the same error condition. Speaking of those last two lines, I could have written them like this instead:

Apply valid formatting for e-mail address Recipient E-mail Address;
Apply valid formatting for e-mail address Sender E-mail Address.

Both forms do the same thing and, in fact, I sort of like the above better but I wanted to showcase that various formats are possible.]

```
Check validation:

begin;

   Recipient E-mail Address must be populated.
   Sender E-mail Address must be populated.
   Recipient E-mail Address - apply valid formatting for e-mail
address;
   Sender E-mail Address-apply valid formatting for e-mail
address;

otherwise;

   Error: "Please enter a valid recipient e-mail address to
process your request."
   Error: "Please enter your e-mail address to process your
request."
   Error: "One or more e-mail addresses are invalid. E-mail
addresses should have the format user@domain.ext
[,user@domain.ext,…] to process your request."

end.
```

[A few notes about the above. If I had put "Your E-mail Address" anywhere in the "source text" (which our current specification does), TestNLP warns:

** You have indicated a rule for "Your E-mail Address" but you have not defined "Your E-mail Address" as specified item. **

In other words, TestNLP recognizes a Recipient E-mail Address and a Sender E-mail Address, but no Your E-mail Address appears and so TestNLP assumes that either you made an error or that it cannot figure out the "model" you tried to construct.

Along the same lines, if I had just put just E-mail Address in the Check validation part above (instead of specifying Recipient and Sender as I did), TestNLP would respond with this:

** You indicate validation for E-mail Address but you do not specify the e-mail address you are validating. Currently your specification mentions two: Recipient E-mail Address and Sender E-mail Address. **

What TestNLP is telling you here is that it is not able to disambiguate from your existing assertions what you mean, but it does recognize that what you are trying to do is valid. It just cannot figure out, from context, what it should do.]

```
repeat with EAF Popup
    Check "[required field arrow]" (called icon) for any field
that is required: icon is left of label of field.
end repeat;
```

[The above code is a bit of a side note but an important one. All of the required fields on the pop-up are supposed to display an arrow graphic. What this does is cycle through any field in the source that that indicates it must be required. But what is "[required field arrow]"? Anytime square brackets are used in quotes, they indicate that something is to be utilized in place of the string. In this case, elsewhere in the source, you would have something like this:

> Definition: required field arrow is a graphic and its URL is http://hsrdb2/images/required_field.gif.

So when that field is tested, it will be checked for the whatever the item in square brackets is. In this case, a check will be made for an icon with the appropriate URL information. Note that since the server may change, you could also do something like this:

> Definition: required field arrow is a graphic and its URL is http://"{ENV}" /images/required_field.gif.

This is just an example since it would have to be defined what {ENV} meant, of course, but what this does is indicate that this should be a placeholder for something. The key here is that text in quotes and in curly braces will be understood by TestNLP as something that it cannot possibly know about and so it rely on the conversion to test logic as handling it.]

1.3 – Generated Elements

From a manual test standpoint, here is a small sample of what gets generated from the above source text (which, remember, is being treated as a specification). Anything with an x means a character and a ___ indicates no data entered.

High-Level Test Conditions:

Recipient E-mail Address: xxx
Sender's Name: xxx
Sender E-mail Address: xxx
Send

Recipient E-mail Address: xxx
Sender's Name: ___
Sender's E-mail Address: xxx
Send

Recipient E-mail Address: x@x.x
Sender's Name: xxx
Sender's E-mail Address: ___
Send

Recipient E-mail Address: ___
Sender's Name: xxx
Sender's E-mail Address: x@x.x
Send

Recipient E-mail Address: ___
Sender's Name: xxx
Sender's E-mail Address: ___
Send

Recipient E-mail Address: x@x.x,x@x.x
Sender's Name: xxx
Sender E-mail Address: x@x.x
Send

Recipient E-mail Address: x@x.x;x@x.x
Sender's Name: xxx
Sender E-mail Address: x@x.x
Send

Recipient E-mail Address: x@x.x
Sender's Name: xxx
Sender E-mail Address: x@x.x,x@x.x

Send

Recipient E-mail Address: x@x.x
Sender's Name: xxx
Sender E-mail Address: x@x.x;x@x.x
Send

Recipient E-mail Address: xx.x
Sender's Name: xxx
Sender E-mail Address: x@x.x
Send

Recipient E-mail Address: x@xx
Sender's Name: xxx
Sender E-mail Address: x@x.x
Send

Recipient E-mail Address: x@x.x
Sender's Name: xxx
Sender E-mail Address: xx.x
Send

Recipient E-mail Address: x@x.x
Sender's Name: xxx
Sender E-mail Address: x@xx
Send

The reason the x's are used in place of actual characters is because TestNLP was not told what data set to use. So what it did was generate conditions that match a pattern based on the specification.

Here is an example of the SilkTest automation that got generated to test those conditions:

```
[-] for each sCondition in rTestConditions
    [ ] EAF.RecipientEMailAddress.SetText(sReceipientEmailAddress)
    [ ] EAF.SenderName.SetText(sSenderName)
    [ ] EAF.SenderEMailAddress.SetText(sSenderEmailAddress)
    [ ] EAF.Send.Click()
```

Here the variables sRecipientEmailAddress, sSenderName, sSenderEmailAddress would be populated with the data that would map to the high-level conditions.

You might wonder, though, how TestNLP "knows" the pattern to put the e-mail data in, particularly given that it clearly generated filler data in lieu of any real data. That is because part of the TestNLP engine's world model has the following built in:

```
Definition: An e-mail address is a text value that follows the
pattern "text@text.text".
```

Procedure: Valid formatting for e-mail address,
Field must have at least one "." after "@";
Field E-mail Address must have only one "@".

You can also use this kind of thing to define how fields are validated.

A key thing to note here about all of this is that if the specification text changed then TestNLP would regenerate the manual test conditions. That, in turn, would force regeneration of the automated test conditions.

What that means is that TestNLP can actually help you with traceability because you can have it put the specification sections in place when you have it generate the manual test conditions. Then if the specification changes, say in terms of what sections are numbered, when you regenerate the manual test conditions, those changes to the numbering are automatically regenerated for you.

Also TestNLP can enforce the use of terms like “must”, “shall”, “is” and so forth to make sure that what is defined are actual requirements.

One thing that probably jumped out at you when reading the Source Text sections and that is that this is all not necessarily quite “natural language” in the most common sense of the word. (From a semantic and linguistic standpoint, I would disagree; from the perception of the most common sense of the word, however, I would agree.) By necessity TestNLP does enforce a certain nomenclature. You can see why, I think, when you consider that it has to take that natural language and force it to generate more structured test conditions and even more structured automated test logic. That said, the rule-engine is flexible. As long as you are consistent, it can generally figure out what you mean. And when it cannot, it will let you know to the best of its ability.

1.4 – Other Uses

One thing I did put TestNLP to use in was with data entry tasks in DAT, particularly in terms of the automation.

I will start with the one I am most proud of. We had an issue that wherein DAT 4 was keeping the state of the MLS Listing and New Construction checkboxes. This was a pain because it meant any listings were potentially categorized incorrectly. However, it is also possible that this functionality can change. (It did twice.) Here is the TestNLP command I used to get around this:

```
When populating MLS or populating New Construction (called
items):
The rule is check MLS for value of Yes or value of No (text - any
case, any formatting);
The rule is check New Construction for value of Yes or value of
No (text - any case, any formatting);
The rule is No is unchecked checkbox;
The rule is Yes is checked checkbox;
Carry out rules for items, otherwise invoke situation handler.
```

I am not going to explain too much about the written out logic here, but what I want to show is how I was able to convert this into SilkTest code and Ruby code.

SilkTest Code (Generated from TestNLP)

```
[-] case MatchStr("MLS", sItem)
    [ ]
    [-] switch Lower("{rRecord.@(sItem)}")
        [-] case "yes"
             [ ] ListingResale.@(sItem).Check()
             [ ]
        [-] case "no"
             [ ] ListingResale.@(sItem).Uncheck()
             [ ]
[ ]
[-] case MatchStr("NewConstruction", sItem)
    [ ]
    [-] switch Lower("{rRecord.@(sItem)}")
        [-] case "yes"
            [ ] ListingResale.@(sItem).Check()
            [ ]
        [-] case "no"
            [ ] ListingResale.@(sItem).Uncheck()
            [ ]
    [ ] ListingResale.Continue.Click()
[-] default
    [-] do
        [ ] EnterData(@("Listing{sType}").@(sItem), rRecord.@(sItem))
    [-] except
        [ ] ExecutionReporter(3000, "Listing", "{sType}", "{sItem}",
"{rRecord.@(sItem)}")
        [ ] continue
```

Ruby Code (Generated from TestNLP)

```
begin
  case recordValue
    when "MLS"
      case values
        when "yes"
          ie.checkbox(:name, 'mls').check()
        when "no"
          ie.checkbox(:name, 'newConstruction').uncheck()
      end
   end
rescue => e
   $logger.log_results(3000, "Listing", sType, sItem,
"#{rRecord('sItem')}")
end
```

I am not going to show the code for each of the things below, but I want to cover a few of the details.

One example is the image buttons for the resale listings in DAT 4. If these are not tagged distinctly enough, SilkTest can have trouble disambiguating which button is meant. So I utilized a bit of my TestNLP on the buttons. Here was the natural language I put in place:

```
When populating ADDL*IMAGE, where * is a number from 1 to 12:
The rule is click first available button with title "Add Image"
in 'OPTIONAL PROPERTY PHOTOS' section;

Carry out rule with no warning if button tag is SR1add or button
tag is SR2add, otherwise invoke situation handler.
```

The key here is that my TestNLP engine was able to work with the browser, so to speak, to figure out what “first available button” meant. That is not something that you can easily incorporate into SilkTest.

The first line is the rule (suffixed with a colon to indicate that what follows is part of the rule; parts of the rule must be separated by semicolons and the whole rule must end with a period). Note the use of “populating” rather than “uploading” as the action word. Right now my rule engine is as simple as possible so that I can merge it with SilkTest. So this means I just check if a field is populated (where “populated” is translated into the appropriate code, in terms of how to populate the field, whether that be clicking a button, adding text to a text field, checking a checkbox, etc).

Interestingly, I cannot just word the part in the last line as such:

...**`if button tag is SR1add or SR2add`**...

What happens with that command is the parser reads the command as a button tag that is called “SR1add or SR2add” instead of recognizing the “or” as a separator of two descriptive clauses.

Another issue came in with populating the pets checkboxes, because SilkTest would get confused if the tag and the actual text of the checkbox differed to any degree (including case). This is mainly because the text has to match a window declaration, which is hard coded. Here is how I used TestNLP to get around that:

```
When populating PETSCSV:
The rule is ignore tags;
The rule is click checkbox field (called field);
Carry out rules for each item in PETSCSV (called text), click the
field that matches text (text - any case, any formatting),
otherwise invoke situation handler.
```

Note that you can string rules together. The parts in parentheses are needed to allow for extra aspects. For example, the “called text” part would be the equivalent of doing something like this in a language like SilkTest:

```
sText = PETSCSV[iIndex];
```

The second parenthesis set, with a hyphen is a way to put a condition on the variable. This would be equivalent to something like this in SilkTest:

```
sText = Lower(RemoveFormatting(PETSCSV[iIndex]));
```

Here, of course, you would have to write the RemoveFormatting() function yourself and use the Lower() function that is built it to SilkTest.

Incidentally, if you go back to my first example in this section (about the MLS/New Construction problem), notice how with the PetsCSV issue I used “carry out rule for each item” while the MLS/New Construction text, I used “carry out rules for items”. This is how I try to simulate looping constructs of various sorts. Also note how the more rules you specify the easier your “carry out” command becomes.

Another example is writing the ability to enter an agent:

```
When asked to enter agent:
Associate agent given by NAME to broker given by ORGNAME.
If ORGNAME cannot be found, invoke situation handler.
```

The first line is what I call a circumstantial command, always ended with a colon. The circumstantial command is what a test script (or user) tried (or is trying) to do. In this case “enter agent.” I have another way of doing this based on participles, which means the command is present participle: “When entering agent:” I am not sure what is the better way to go yet. That first line is an “instruction” or “rule”, basically, in my logic.

An assertion statement would be the second and third lines. The verb “Associate” has a defined meaning based on the context. “[Associate] [agent to broker]” is how that is initially broken up via the parser. (If it were a listing being entered, it would be [Associate] [agent to listing] and [Associate] [broker to listing].) The semantics of [agent to broker] are analyzed and the appropriate “Associate” method is referenced. This method is where the implementation logic is handled: i.e., the actual actions that have to take place in order to do this in DAT.

There is further analysis here in that “given by” is read as a separator that indicates a linkage. So the phrase, from a natural language processing viewpoint, is broken up as:

```
[ [Associate] [agent {given by}[NAME]]
{to}
[broker {given by}[ORGNAME]] ]
```

Likewise “If ... ,” is a conditional assertion statement and the comma is currently necessary to indicate what should happen after the assertion. I suppose technically the comma is not necessary because as long as a verb form is encountered after the conditional assertion statement opener (in this case, “invoke”) that will be treated as the second part of the command to implement. Anyway, here [cannot be found] is actually broken out into logic that determines, via the context of the previous assertion (“Associate ...”) what “cannot be found” means. This means I could say instead:

`If ORGNAME is not found`...

or

`If ORGNAME was not found`...

The syntactic and semantic elements of the TestNLP engine, if sufficiently learned, will handle variations like this. That said, I could see this being standardized a bit more.

Note also that NAME and ORGNAME can be thought of as a variables that refer to whatever the name or organization name being dealt with is. That said, NAME and ORGNAME are actually more like macros that get expanded to deal with some logic of their own, such as (1) where to get the NAME or ORGNAME and (2) how to validate the NAME or ORGNAME.

A bit of a side note to this but I find that I have to avoid syllepsis like this:

`Associate agent given by NAME and associate broker given by ORGNAME.`

Here the “and” couples together verbs rather than nouns and that is (for some reason) a problem for the parsing engine.

This seems like a good place to end as I think the examples have served their purpose.