Model-Based Testing for Data Centric Products

Maig Worel
Microsoft Corporation
mworel@microsoft.com

Abstract

Data centric products such as language and language-based logic products can pose an overwhelming test problem. Here is an approach to begin defining an omniscient test suite to attack search engine bugs using model-based testing theories in the real world.

Content

What's so difficult about testing Data Centric products?

For the purpose of simplicity, let us consider an email application. Most email applications do a few, very simple operations – and while it is easy to test the functionality, most of the bugs are found in a mountain of data. Like most email applications, ModelMail (our app) creates mail, sends it, and retrieves and displays email. This behavior is unlikely to change over the course of the product. However, some of the rules about how different types of email, such as email with different languages or other attributes, are HIGHLY likely to change (or should be) over the lifetime of ModelMail, as developers fine-tune their algorithms.

A popular test model for ModelMail would tends towards throwing large bodies of testers at the search engines, clearly defining a few right answers to a few known questions, and hoping that any other questions over any other body of data will follow the same pattern of behavior.

The largest issues are in the Data and the Product itself:

Issues with Data:

> The data to test ModelMail must closely, if not completely, resemble real world data. Yet, real world data is messy- it has so many words, so many grammatical forms, so many shapes.
>
> The test suites must run tests (questions) that have known answers in order to determine if the product is running correctly. However, if only those questions are tested, the product quickly becomes subject to the "Pesticide Paradox"- you've found all the bugs for the questions you have defined, but who knows what evil lurks beyond those well-traveled paths?
>
> Searching through large test data sets (say, 200,000 documents) is EXTREMELY time consuming, and is rarely automated. When automated, the relevance of the documents to the questions is often as generic as a very bad search engine. In other words, the test suite only knows that this document has words in it that are also contained in the question posed, and not whether it is automated. When annotated by hand, not only can the annotation take months, but also it is subject to human error.

Issues with Email Applications:

> Even in a well-monitored, highly documented product, there can be a shifting landscape of features and user sets. What kinds of character sets are supported? What kind of relationship between words is important to your application? What are words? What words are important? What does a good data set for an email application look like?

Modeling the product, so as to change the tests rapidly, and modeling the data, so as to change the domain rapidly, will keep you in control and on top of your testing!

Benefits of Modeling

By abstracting the behavior of the product and the attributes of the data, you can create a test suite that requires very little maintenance. This is desirable because in a grammatical product, behavior in terms of how a test is run tends to change very rarely (data in, file out) but the behavior changes very rapidly (one word receives one answer one time, a different answer the next time).

By splitting the unchanging aspects in the code, and the changing aspects in an easily manipulated ASCII file, the maintenance time is greatly reduced. This is the difference between "hard-coded" and "dynamic" programming- to create a test system whose data is easily changed while the essential behaviors are not.

Of course, this is true of any test suite program. In modeling, the test cases are further abstracted – down to the essential interesting information in what the test case consists of.

Currently, typical test cases might be:
The user wants to find out more about Grammatical Modeling. The user enters into the UI a subject for an email:
Open a file and send a mail about "Models" and close the program. Did the mail get sent?
Open a file and send a mail about "Testing Models" and close the program. Did the mail get sent?
Open a file and send a mail about "RE: pages about Modeling Tests" and close the program. Did the mail get sent?

Results: Several different email items have been. The tester then attempts to send the mail, and observes the results. Sometimes the result for one test are saved and used to automate the next test run by comparing the last results for a subject with the current result for the same subject.

Without modeling, these cases are very tedious. A tester must look at each result and evaluate the quality of the responses. To truly eliminate all bugs, testers for ModelMail might have to try at every subject in the world!

With modeling, the test cases might look like this:

If we abstract the first three tests into a grammatical model of <subject>, where subject is one or more words that might be a subject, we can mix an match many different words (even languages) to create many tests, very quickly. You can see how the model of <subject> can be used to create the third test case – where the model might be <subject> => <pre-subject><phrase>.

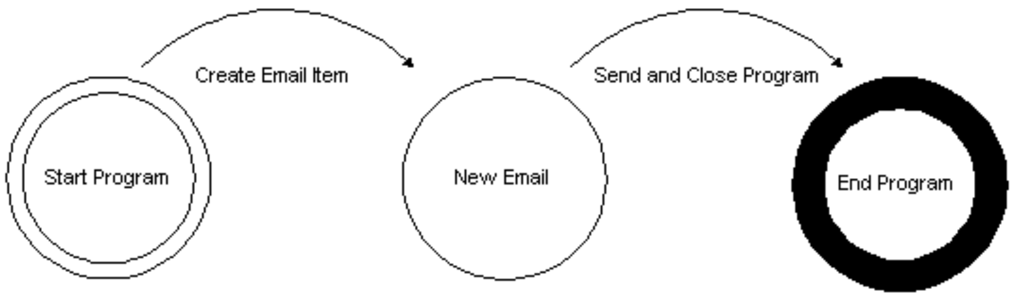
<subject> => <pre-subject><phrase>
<pre-subject> => RE: | FW: | <nothing>
<phrase>= "pages about" <verb><noun> | <verb><noun>
<noun> => Tests | Models
<verb> => Testing | Modeling

By automating this model, testers can quickly generate far more than the original cases.

Using Grammatical Modeling quickly gets testing started, and allows for rapid expansion of testing suites. All of this is automated, so the tester spends more time on key product issues, rather than developing and running test cases. This is done by use of a model, and a testing tool that applies this model to the product.

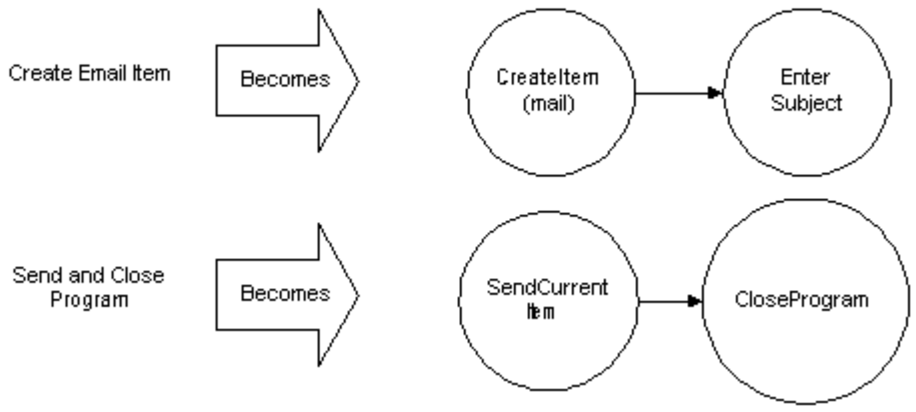
Modeling an Email Product

Both the product and the test data are modeled. Starting with existing documentation – and note that without any documentation, preliminary modeling can still be begun - the simplest model of an email application would be the input, and the output.

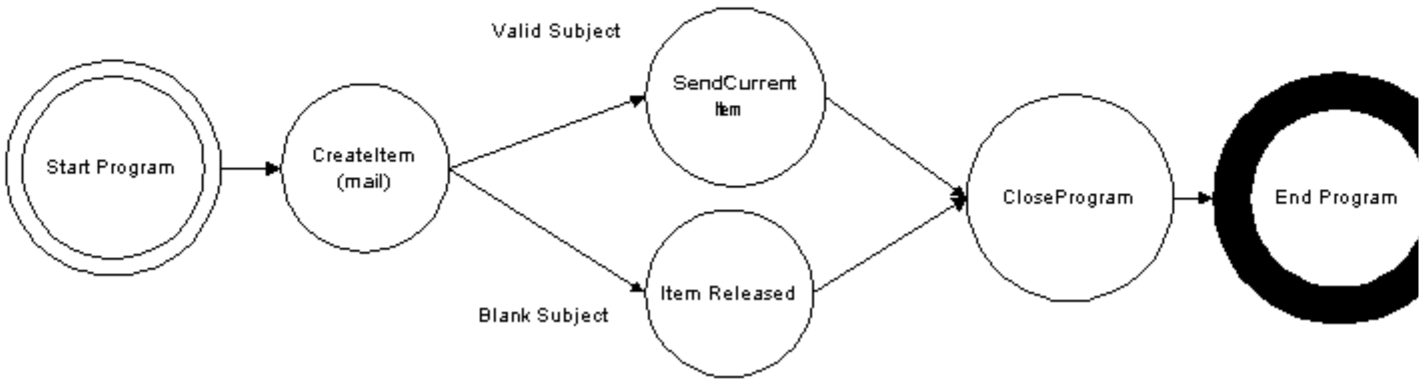


There are two states- pre- (before email is created) and post- (when email is sent) input. A simple robot out that combines different characters in different ways, sizes and shapes just to see that things are returned can test this model. This is the basic building block of your Model-based testing tool. While this simple tool pounds out boundary cases and nonsense words in the background, there is time to model the other rules to follow, and the document sets these rules will produce.

Now we begin to break down the inputs into sub graphs and the end states into sub graphs as well.



This may come a shock to some of you, but - not all of those rules will be documented. Even developers don't always REALLY know what the product will do- that's why God created testers. So now, to further model your product, get your hands on any documentation you CAN find, and begin exploring into the great unknownFor instance, say you model that all good input in the subject will produce email that can be sent. One of the inputs you apply is an empty subject. ModelMail returns to you no error, but no email is created. Now we know that another rule applies to our model: subjects that are blank don't create mail (you suspect that a blank email is the true model here). So now we add that to the model.



It sure doesn't take long to model most of the functionality for our email app. But, how do we keep track of what we have created, and what state we are currently in? We model the documents, and we model the inputs, which we will call "tests" and keep track of our model as the test suite runs. Now we turn to modeling the important part: the data itself.

Modeling the data – Grammatical Models

We have already started to do this type of modeling above, with results. ModelMail has two forms of data: the email items, which are the "domain", and the test, or "actions", which are like a traversal or path over the domain. When you apply a test to an email item, the response is a new item state – a new point on the graph of the test. It's geometry, all over again. Who said those math classes would never be useful?

We can begin modeling with the tasks, and generate items that will fulfill those tasks, or start modeling with items, and generate tasks that will "create" particular item states. Usually, products have specific items sets that the product MUST work with, so let's start there. ModelMail is a email app, so it will need to create many different forms of email texts, in many languages, with perhaps some html inside and web page references. We can add other interesting cases to the model later.

Here's how the model begins:

Open a file and send a mail about "Models" and close the program. Did the mail get sent?

Open a file and send a mail about "Testing Models" and close the program. Did the mail get sent?

Open a file and send a mail about "RE: pages about Modeling Tests" and close the program. Did the mail get sent?

If we abstract the first three tests into a grammatical model of <subject>, where subject is one or more words that might be a subject, we can mix an match many different words (even languages) to create many tests, very quickly. You can see how the model of <subject> can be used to create the third test case – where the model might be <subject> => <pre-subject><phrase>.

Now, let's break down this model into more specific – and useful – rules, and create some instances.

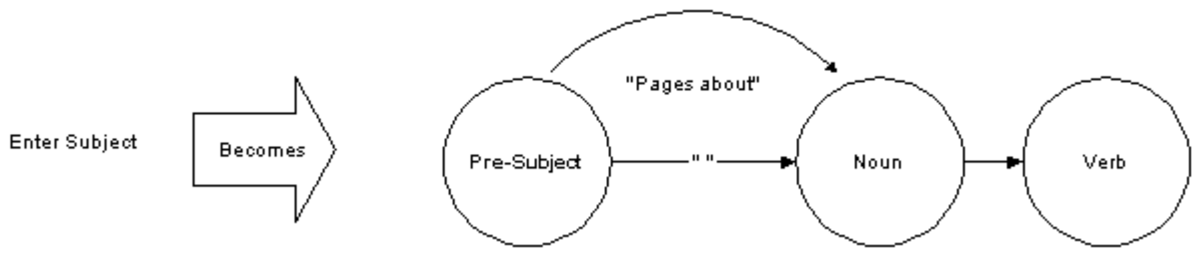
<subject> => <pre-subject><phrase>

<pre-subject> => RE: | FW: | <nothing>

<phrase>= "pages about" <verb><noun> | <verb><noun>

<noun> => Tests | Models

<verb> => Testing | Modeling



You can think of a document or an email object as a graphic traversal of the document domain. There is the start state (a blank item with no attributes filled in) and the end state (wherever you leave off testing). In between, the document may have words added, which may have punctuation (thus adding a sentence state), a carriage return (the creation of a next paragraph state), a hard or soft page break (the creation of a new page state) etc. Some of these words may be important. The distance and order of these words may be important. On web pages there may be forms like the <head></head> section, and the <body></body> sections. Many documents have these types of forms, and you can create a model for these forms.

Here is a simple one for a body about modeling:

```
<HtmlFile> = <start html file> <Body> <end html file>
                                        //for brevity,  let's skip to sentences in the body
<Sentences>=<Sentence>|<Sentences><Sentence>
<Sentence>=<Simple Sentence>|<Complex Sentence>|nothing
<Simple Sentence>=<Subject><Verb Phrase>.
```

<Subject>=Model(ing) | Test(ing) | Pre-Oracling
<Verb Phrase>=rules | is exciting | is fascinating.
<Complex Sentence>= error – not implemented

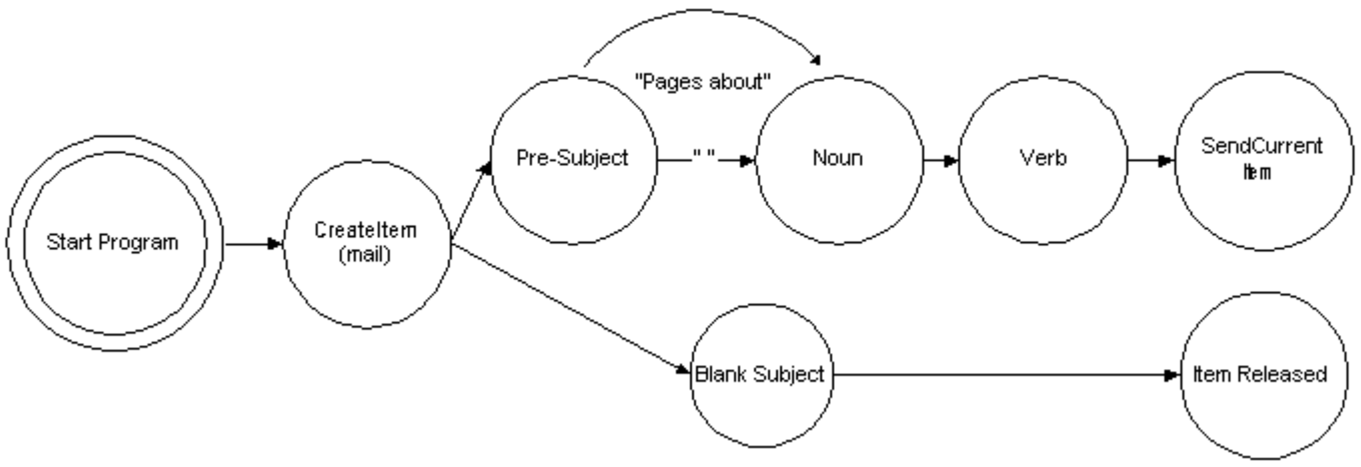
Extra Rules for model (these tend to be coded rather than generated because they are true at all levels):
Case (capitalizing) may be unimportant.

The Document Model

This model (and any others created) is stored in separate, parseable data source, such as a text file. How you choose to implement this is the key to maintaining flexibility. Be sure to choose a method that leaves it easy to change the model.

So now you create an application, which creates random combinations of the above models in some prescribed order using your Domain Model data source. For instance, I may tell my app to create 5 email body items of 3 – 5 sentences of various length (up to … say 4 combinations) and save the information out to some large data store. This application reads the Grammatical Model, and randomly chooses substitutions at each level until it reaches an ending point.

For instance: the first level it reads is <File>
The only substitution is <start file><Body><end file>
So, it opens a file, puts an end of file marker on it, and then needs to fill in the <Body>.
    <Body>
The only substitution for <Body> is <Sentences>
    <Sentences>
It randomly chooses <Sentences><Sentence> to substitute for <Sentences>
    <Sentences><Sentence>
It randomly chooses <Sentences><Sentence> to substitute for <Sentences>
    <Sentences><Sentence><Sentence>
It randomly chooses <Sentence> to substitute for <Sentences>
    <Sentence><Sentence><Sentence>
It randomly chooses to substitute <Simple Sentence> for <Sentence>
    <Simple Sentence><Sentence><Sentence>
The only substitution for <Simple Sentence> is <Subject><Verb Phrase>.
    <Subject><Verb Phrase>.<Sentence><Sentence>
It randomly chooses to substitute "Test" for <Subject>
    Test<Verb Phrase>.<Sentence><Sentence>
And it is done with choices, so it moves to the next section that is not yet resolved.
It randomly chooses to substitute "rules" for <Verb Phrase>
    Test rules. <Sentence><Sentence>
And so on through the Grammatical Model.



Now you have several email items with bodies like this:
Insert html files here
File 1. Test rules. Modeling is exciting. Pre-Oracling rules.
File 2. <empty>
File 3. Pre-Oracling is exciting. Testing is exciting. Modeling is exciting.
File 4. Modeling rules. Modeling is exciting. Modeling rules. Modeling rules.
File 5. Testing rules. Pre-Oracling rules. Testing is fascinating.

And a data store like this:

FileName/Location	Test	Model	Pre-Oracle	Rule	Excite	Fascinate
File 1	1	3	6	2,7	5	0
File 2	0	0	0	0	0	0
File 3	4	0	1	0	3,6	0
File 4	0	1,3,6,8	0	2,7,9	5	0
File 5	1,5	0	3	2,4	0	7

The Document Set Instance

Some things to observe about this data store: I have stored the minimum information needed – but I am anticipating here that my email app will be able to do text searches, and I will want to test that. I don't care about "is" and "ing" for verbs in my model, so I don't store them as column names, and I store the "stem" of the verb as my word count. File 1, sentence 2 I am storing "is exciting" as "exciting" – but another product may want the word count to start with the "is".
In this product word count, rather than letter count, is the valuable information, so that is the number stored. If I thought that perhaps sentence structure might be important, I might store it more like this:

FileName/Location	Test	Model	Pre-Oracle	Rule	Excite	Fascinate
File 1	1:1	2:1	3:1	1:2,3:2	2:3	0
File 2	0	0	0	0	0	0
…						

Granted, these do not look like good sentences. But if the model above is correct, these will work the same as the files generated above. Isn't modeling fun? Yes, you can write your generator to produce only good sentences, and turn them off with a switch to test error cases. But, as testers – we like to make sure we cover these strange cases as well. Just to be sure.

Tracking your tests – State maintenance

You have this beautiful picture of your product (the ModelMail model), and the domain that it can create cases from in the Grammatical model. Now what?

Write a tool to use the two datastores you have created to run your tests against the documents. You know everything (for now - you think) about the things that are important in a document.

Heres what the tool does: as you run your tests against the product, run the SAME test against your Data store. This is the key to testing the validity of your tests, automatically. Use the test model to cause changes in items in the Data Store.

For example:
I start with a blank email app, and a blank store.
I create an email with the subject "Modeling" and the body "Modeling rules."
    I also copy that information into the Data Store, and this row is marked "new mail"
I send the email
    I mark the row "sent mail" in the Data Store.
I check that the mail is in the sent folder, and that the information is the same as I have stored in the Data Store.
    Note: at this point, I can also check the entire email app against every item I think should be in the store.
I run the next test… and keep state in my store…

If we see different behaviors than what our model predicts, they may be bugs – or they may be an error in the model. As we run these tests we develop a more clear view of the behavior of the product itself. In this case, we see that the forms of grammar may be the same for this and for some other email items. Using the Grammatical Model, we can quickly compare the items in question to see where they differ.

We might direct our test generation tool to generate more email with the same pattern, to see if they also generate a discrepancy. If there is, we found a bug. If not, we found a hole in our model.

An iterative process

I know I've had a few thoughts while writing this about changes I want to make to the model. As you run tests, and discover new behaviors, you will also. The beauty of creating tests in this manner is that the tests themselves are not hard coded – only the method of submitting the tests, gathering the results, and comparing them to the oracle is. We can quickly add new synonyms to the query model, and new patterns to the document model.

If in addition, you have rules about ranking, such as "emails from my boss should appear higher in the UI" then the same email create and send tests might return different orders to be added into the model.

If I want to run queries of a particular grammar, I can use the Grammatical Model to filter in or out particular grammatical forms. For instance, I may find out that the form for breaking down email with certain characters may be broken. So I only run forms for which those characters are not used.

Conclusion

Language testing can be modeled in a realistic time frame. Modeling the behavior of the product and attributes of the data give testers the flexibility and focus to gain greater coverage and bug discovery in the product.

Modeling language based products can be very simple to start, and made as complex as time allows. The greater the detail of the model, the better coverage of code, and the higher likelihood of discovering bugs.

Modeling these products gives you the flexibility to:

- Change quickly as the product or user focus changes
- Focus tests to particular areas
- Become more detailed as time allows

References

Boris Bezier. *Software Testing Techniques*, New York: Van Nostrand Reinhold, 1984

Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software*,2/e Boston: International Thompson Computer Press, 1993

Brian Marick *The Craft of Software Testing*, New Jersey: PTR Prentice Hall, 1995

Peter M. Maurer, *The Design and Implementation of a Grammar-Based Data Generator*, Tampa: University of South Florida, unknown

Peter M. Maurer, *The DLG Manual*, Tampa: University of South Florida, unknown

Harry Robinson, various papers on "Model-Based Testing", Redmond: Microsoft, 1999, 2000