Model Based Testing Of Web Applications

Change is a constant - sounds like an oxymoron but we in the test fraternity all accept it to be true. Change is inevitable, of course, and in a rapid development environment such as software development, change and constant change is an all too common occurrence. As prudent leaders in the testing profession we make allowance for change. It does not change the facts, however; even a minor requirement or software change can mean major upheaval in test preparedness, schedules and resource needs.

How do we limit the impact of change, improve our timeliness and responsiveness as well as accommodate the ever-increasing demands placed upon our time, talent and resources?

In response to the challenges posed by change, at DDI Health, we have implemented what we believe to be a highly innovative and yet practical, if little practiced or understood technology. Model Based Testing (MBT).

Hani Achkar

Email: hani.achkar@yahoo.com.au

Hani currently holds the position of Test Manager with DDI Health, a Perth Western Australia company specializing in e-health, offering enterprise IT services, PACS for radiology, and clinical IT solutions across the diagnostic and general healthcare market.

Previous to his role with DDI Health Hani has held the position of Test and Reliability Engineering Manager with Xtralis, Test Consultant with IBM GSA within the Test Centre of Excellence, Test Manager with Australian Defense Industries and Principal Test Engineer with Adacel Technologies.

Hani also holds US and UK patents in Model Based Testing of Embedded Systems.

What Are Web Applications?

Web applications are those applications that communicate with clients via web protocols such as HTTP, WSDL or SOAP for example. Their preponderance and ubiquity are direct results of the pervasiveness of the World Wide Web as well as the increasing utilization by developers of web browser's to host their applications. Web browsers provide the developer with a unique opportunity to exploit the web browser's inherent capability to relieve them of the burden of having to cater for the various operating systems or hardware platforms available to the intended users of the application.

Today web applications range the gamut from dating sites to gambling sites, ecommerce, on-line banking, airline bookings, and corporate websites right through to applications that are not meant for

general access but never the less use web browsers to be hosted. Of this type there are many and the number is growing. At DDI Health many of our main products are web applications, to name a few; FIT (Filmless Image Technology) a filmless digital radiological imaging solution; Pathology Web Portal a portal that permits over the web secure access for medical practitioners to view patient pathology reports; DDI's RICS (pathology Request Image Capture System). Many more examples may be cited. Suffice it to say that the realm of web applications is very rich and growing.

The Difficulty of Testing Web Applications

Developing web applications essentially is no different to any other software application; you start with ideas, mock-ups and requirements and in a managed fashion work towards a working solution. Testing web applications should be just as "routine" a task. You start with requirements and mock-ups, decide on the priorities for testing, design the tests, execute the tests and report on the outcomes. You go through this cycle two, three, or more times until you reach your quality objectives (if you are lucky) or you simply run out of time.

That is an ideal, the reality can be much different and more often than not is. All of a sudden what you thought were requirements agreed and understood begin to change, and not just in the ones or two's but eventually in avalanches as the end users or major stakeholders begin to question and challenge original concepts and thoughts.

What was originally agreed would be a link now changes to a button (fair enough). Then someone decides that alteration to signing in is required for legal reasons. Now signing or logging in requires acknowledgement of legal terms and conditions and further requires a check box feature that must be checked before the "Login" button is enabled as absolute proof of acknowledgment. OK, that is good, we can handle that. Another change arrives soon after; now it is required that a unique error message for each error type that may arise be displayed instead of the general error message that simply alerted the user to errors in their entries. The error messages now will be unique to each entry field and will uniquely reflect the type of error needing correction, it shall be presented in red, 12 point Calibri font with the field in question highlighted by a red background– we can test for that it just means a lot more combinations than we figured on but we will attend to that, just more tests to design and execute. There is nothing strange in any of this; these are the day to day challenges that face a tester, the test schedule and undermine the test plan.

Web applications are a strange beast indeed; they are a hybrid between a web site and a regular application; they provide feature rich capabilities to potentially millions simultaneously. Web applications provide the user with an extremely rich experience which is way beyond what mere web sites alone can offer. Consider an ecommerce web application, a Florist web site – say Interflora™. You as a user are at liberty of storing your billing and shipping details, further you may specify multiple payment methods which may include personal and corporate credit cards. You may wish to have multiple orders placed on a regular basis to different recipients. This suggest that within the Interflora™ web application a highly sophisticated address book and order processing application is built in. Consider the complexity of testing robustly just this feature. Superimpose on this problem all the

issues related to cross browser compatibility, aesthetics and usability issues and you begin to appreciate the magnitude of the test problem.

Get it wrong, miss something, or fail to conduct adequate progression and regression testing and real problems may occur real fast out in production. The issue is that defects that are missed in testing, upon release of the web application, are immediately available to and are at the mercy of a very large, unforgiving and vocal user base. From thousands to millions of users will be exposed to and use the web applications in ways that a test department of five, ten, twenty or a hundred may not be able to design and execute enough tests enough times for. Simply the number of potential usage patterns immediately and simultaneously imposed on the web application will exceed the number of patterns derived by the test team in the project schedule available to them. It can be an overwhelming experience.

Adequate testing relies upon adequate test generation, execution and maintenance. Applications that have exposure to very large user bases are the most challenging problems facing software test organizations. These organizations must contend with software complexity which is ever increasing whilst test schedules are ever shrinking. In a standalone application if it goes wrong the cries from the field won't be heard for a while – short while, but a while none the less; they may even arrive in a steady stream. On the other hand web applications gone wrong in the wild have the potential and capacity to release a deluge of complaint and dissatisfaction almost immediately, the worst of which is revenue loss to the stakeholders.

Automating Testing With Scripts

A response to the dilemma posed in the previous section is test automation through scripting. This is a feasible approach and many organizations have made significant investments in test automation staff, licensed tools and frameworks. Other organizations have decided to employ open source tools that support test automation as opposed to licensed software.

Most test automation tools, open source or proprietary licensed, employ one of two techniques: record/replay augmented with user scripting; or outright scripting without resorting to record/replay techniques. Such automation techniques and tools do work and depending on the proficiency of the practitioners can and do work well and have been applied very successfully. However, they do have an Achilles heel, time to automate. The reason is simple before a test can be automated it must first be created, validated, executed, then scripted and debugged.

These are reasonable and necessary steps, albeit time consuming. Additionally, given the time required to adequately undertake these steps, it is clear and evident that the benefits of test automation under these circumstances could only ever truly be realized in regression testing – it is unlikely that during progression testing that these automation steps can be adequately accommodated.

Any form of automation must be an improvement to manual test execution. Further, the number of tests that can be automated in the time available generally cannot hope to represent the totality of the tests created for manual execution and therefore are a thoughtfully selected subset. A potential issue with test scripts is that unless they are highly parameterized they will suffer the same fate as manual

test cases. That is they will suffer from what Beizer[8] calls the "pesticide paradox" whereby tests become increasingly less and less useful at catching defects, because the defects they were designed to catch have been caught and fixed.

A further issue is that mostly the test scripts are static. How can test automation scripts be static? Consider the source of the script. Either it was generated during a record/replay session and possibly further elaborated through additional scripting, or originated from a manually created and individually crafted design encoded in a script. The script is, from the time of its creation, fixed until altered by its creator. That is fine, as long as the behavior it was created to dynamically exercise does not change or change too often or have flow on effects to other related scripts.

However, reality is that requirements change as we have already mentioned and consequently behavior changes and this happens far too often for us to ever feel comfortable or be complacent. Change is inevitable.

The following quote from (Mosley, 1997)[1] is somewhat humorous (who thought testing was easy?) although telling of the risk of automation scripts:

> *"I worked with one Visual Basic programmer who had previously constructed and maintained an automated suite of test cases for a client-server system. The problem he encountered was that the features were never frozen and he was constantly updating the test scripts to keep up with all of the changes. Of course, he never caught up. He swears he will never do testing again!"*

The experience related above is not uncommon or unusual. As specified behavior changes test scripts need to be changed. For this to happen it is prudent to first amend the test design, re-execute it to ensure it is sound, and then release the script for general application. This time consuming activity needs to be undertaken on every occasion of a change of the behavior the test scripts were created to exercise. If it was only one script that we had to worry about there would be no issue. However, when we are dealing with applications that at their source started from several hundred requirements which go through a rapid series of amendments and additions, you can easily see the monumental task that arises to create, maintain and update test scripts. Very quickly the task of maintenance may overtake the task of creation or execution. Many organizations eventually find that keeping up is impossible and automation slowly vanishes, or they have to create dedicated teams to maintain these scripts – that is added overhead and cost.

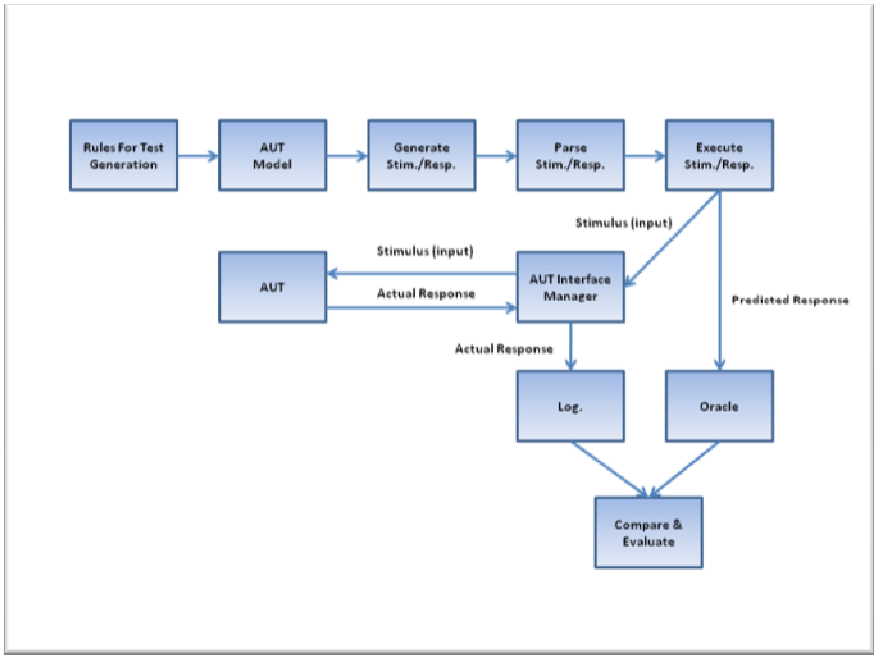
A solution, I believe, to this problem of creating and maintaining test scripts, especially when considering the ever increasing demands on testing and the intense complexity of web applications is Model Based Testing.

Model Based Testing

In the domain of model based testing it is generally understood that the model is an abstraction or simplification of the behavior of the application to be tested. The model is captured in a machine readable format with the sole purpose of acting as both test sequence (trace) generator and oracle.

Model based testing offers essentially two modes of operation or execution: On-the-fly and off-line execution. Typically off-line execution requires the oracle aspect as the test sequences generated through and retained by the model for future execution need to be paired with predictions or estimates of the expected results. A mechanism to compare the actual application response to the model predicted response needs to coordinate the recording of actual results and the evaluation against the predicted results. Typically this arrangement has architecture similar to Figure 1:

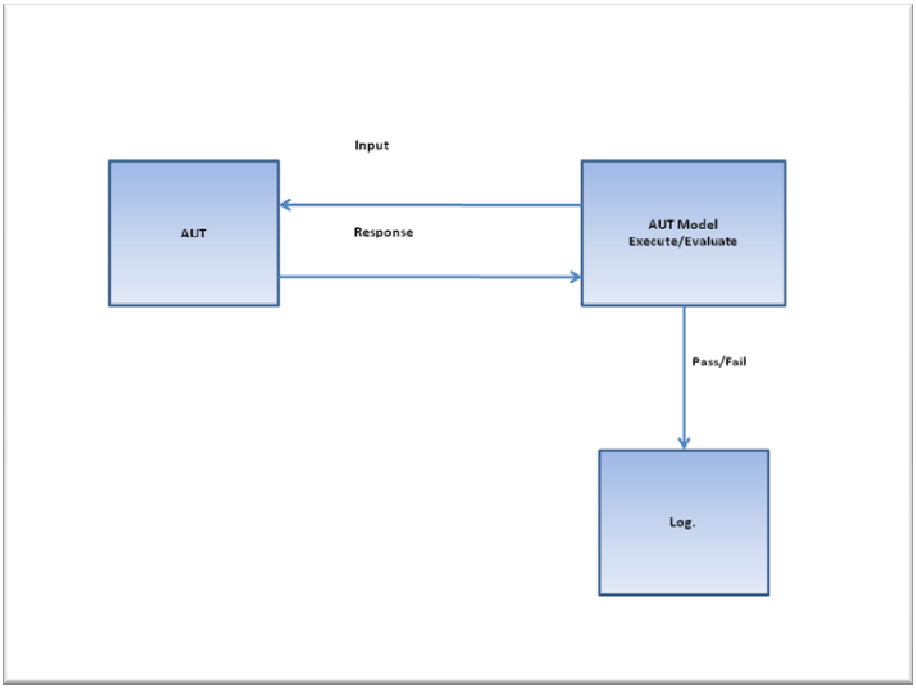
Figure 1 Offline Model Based Test Architecture



What Figure 1is saying is that for a given known modeled behavior and a specified input the model can **automatically** (programmatically) calculate or **predict** the expected response for the application it represents providing for automatic exercising of the behavior and comparing actual response to predicted response.

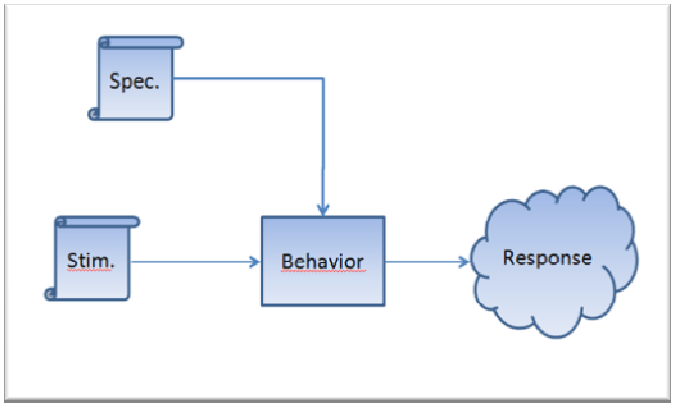
An improvement to this general approach to model based testing is the on-the-fly execution of Model Based Test. In this mode the model generates the test sequences (traces) whilst dynamically interacting with the application under test (AUT) – this in a way mimics user interaction with the AUT. The model program has been so constructed so as to dynamically interact with the AUT, provide inputs, check the responses and attributes returned by the AUT for the inputs and compares those to expectation held by the model. This for me is a simpler and more appealing Model Based Test approach to the off-line approach.

Figure 2 On-The-Fly Model Based Test



When you consider what the automated model is actually doing it is not so far removed from the basics of all testing. Consider the most basic and primitive Black Box test design, see Figure 3 Black Box Test View below. The tester considers the specification of the application under test. They consider the behavior required by the specifications relevant to their interest and then consider the input domain that will exercise this behavior – that is the input or stimulus. Based on the tester's appreciation of the specification and the expected behavior the tester calculates the response of the system to the input or stimulus they provide – this then is the response or result.

Figure 3 Black Box Test View



The tester then, based on their model of the behavior of interest, exercises the application under test with a known input for which the behavior has a predictable and predicted response. Testing then as discussed here is consistent with **exercising** a known or expected behavior; note the emphasis on

exercising which denotes a sense of motion or dynamics. This view is supported by many definitions used for software testing.

"Software Testing consists of the dynamic verification of the behavior of a program" [1]

Thus when considering Model Based Testing in particular, we may rightly state that:

Model Based Testing is the automated dynamic verification of software behavior under programmed model control".

Finite State Machine Model Based Testing

There are many approaches to proposing a model for the purpose of testing. However, in this paper we are only interested in models which define the **behavior** of an application in terms of its States and those *actions* that change its State. Such models are defined as Finite State Machines (FSM).

What do we mean by Finite in Finite State machine? Finite simply indicates the limited number of States of the AUT. If you prefer to review a more formal definition of FSM you may visit NIST[3] online which provides one of the better definitions available. A FSM model then is represented as a graph or state chart comprised of a limited (finite) number of nodes (States) interconnected via directed edges (Transitions – actions).

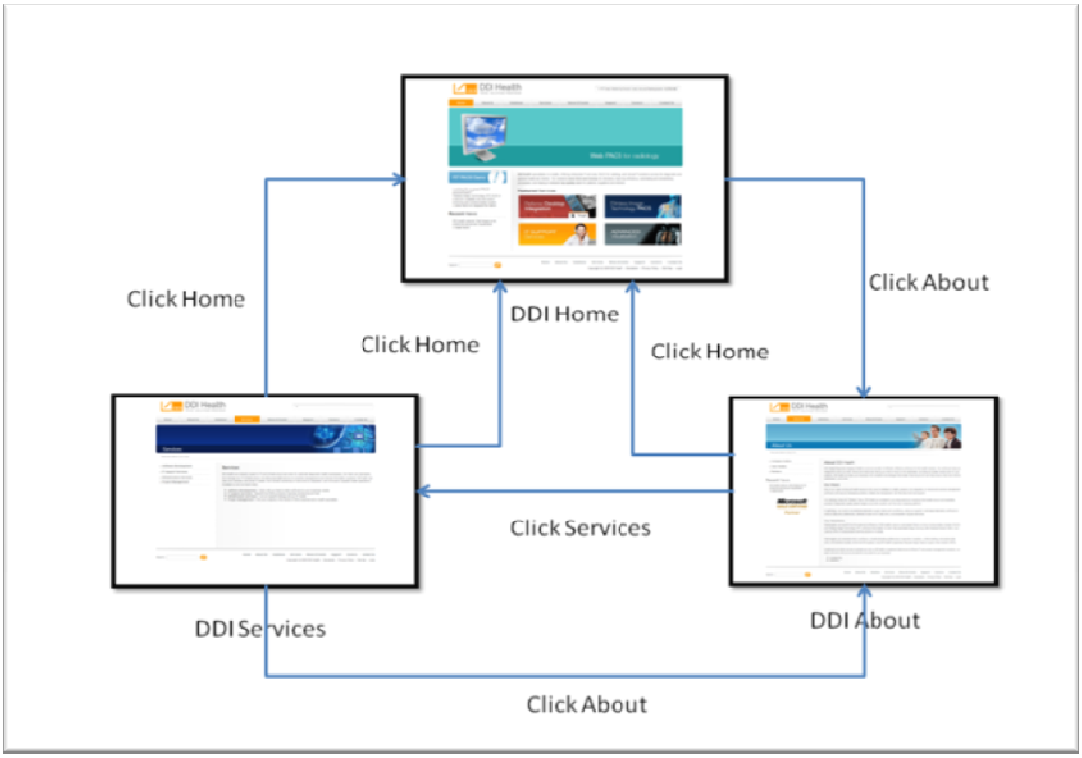
When an FSM model is declared in a machine readable fashion and coupled with an automatic means of being navigated under the control of traversal algorithms and provided with an interface infrastructure to interact with the AUT, it is a very powerful automated test tool. Such tools exist and one in particular (TestOptimal™) will be discussed later.

How then to construct FSM models of web applications? When considering a web application we can easily propose a model of the application in terms of FSM's by following a few simple rules:

1. Each page of the application can be equated with a State of the application (in a black box sense).
2. Each tab of each page can be considered as a sub-State of that page.
3. Every action that alters or changes the page of the application in a way that you care about results in a State change
4. Every action taken can be equated with a Transition for the purposes of the model.

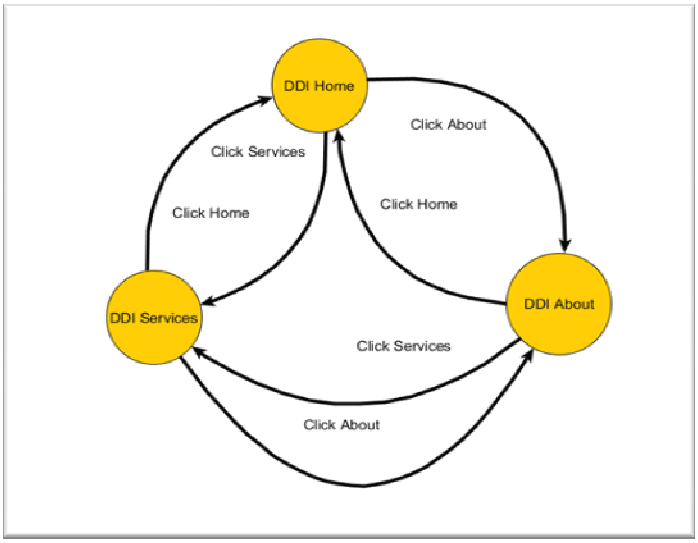
Consider a very basic model that looks at a trivial view of the DDI Health website [4]. We are interested in looking at the behavior of the web site only as far as the Home Page, About Page and Services Page are concerned and how we may navigate between them in the manner depicted. In my mind is an abstraction of the situation as follows in Figure 4:

Figure 4 Abstraction of DDI Health Website



Now based on the rules defined earlier for proposing a FSM for a particular application of interest we may continue to Figure 5:

Figure 5 FSM of DDI Health Website

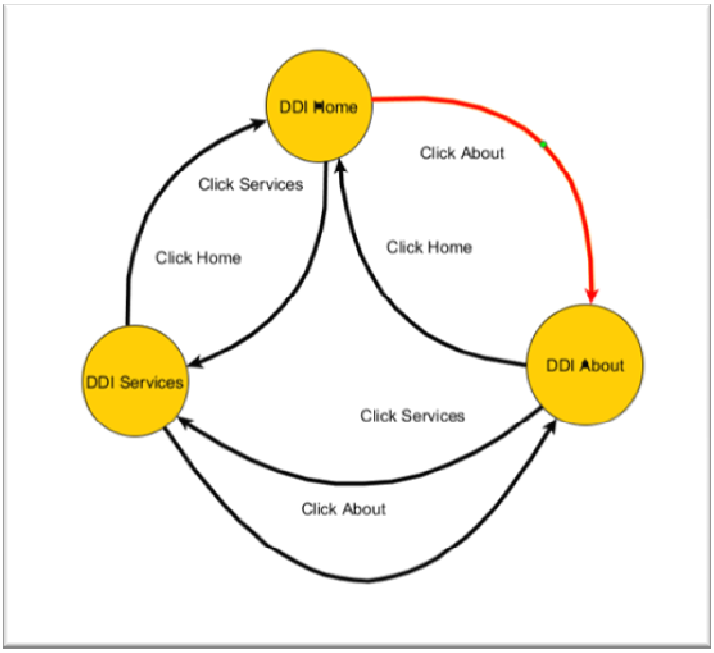


We have transformed our abstraction of the website to an FSM representation which now allows us to visualize the behavioral interaction in a simplified manner with less distracting detail.

An FSM, as can readily be seen, provides for a simplified abstraction of reality where the behavior of the reality we are addressing is represented by States (pages/tabs) and Transitions (actions that case movement) between States, these are the actions that move us between the pages.

If we were using the FSM as depicted in Figure 5 to identify tests that we could execute we could for example color a single Transition as a test, an example of which is depicted by Figure 6.

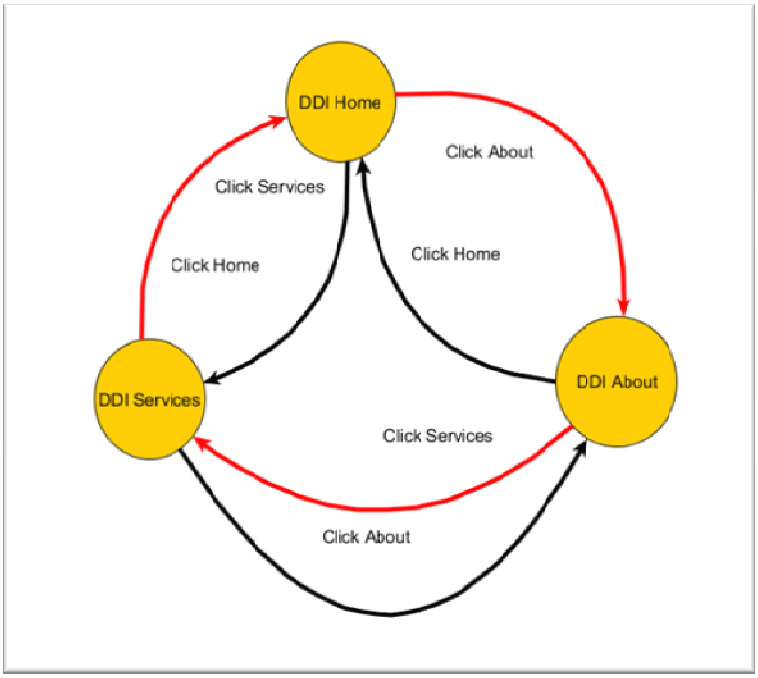
Figure 6 A Single Transition Through FSM As Test



Or, more commonly and potentially more meaningfully, paths through the FSM can be colored as the test sequence as depicted by Figure 7. Of course when you consider even this trivial example you can already imagine the large number of paths that you could identify. Try it as an exercise. Every test is a valuable test and is capable of potentially identifying defects.

It has been shown in several technical papers and publications how exploiting path coverage in FSM's has detected difficult-to-find bugs.

Figure 7 Path Through FSM As Test Sequence



For the simple FSM discussed here there are in fact only 6 transitions and if we only focus our coverage on transition coverage, then we potentially only have six tests to execute. However, if we select path as our coverage metric it is clear very many more paths may be exercised.

Thus testing an application is akin to traversing a path through the graph of the model. Utilizing techniques derived from graph theory permits us to easily and efficiently use the behavioral information captured in the model to generate new and useful test sequences.

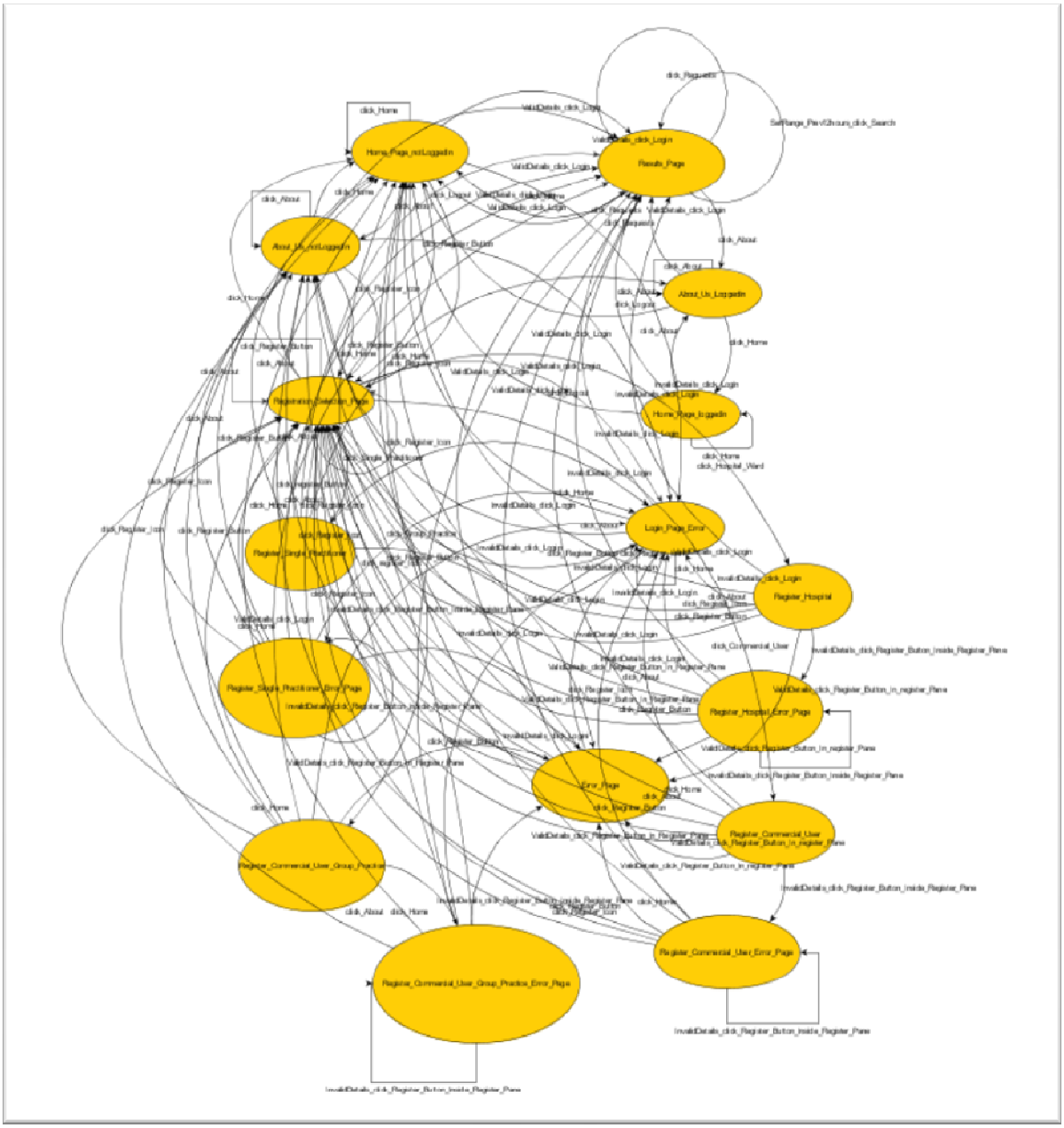
Thus:

- As the model changes in response to changes in the underlying requirements new traversals are able to be generated very quickly resulting in new test sequences thus avoiding the "pesticide paradox" discussed earlier
- By feeding new sets of data to the model in a manner consistent with Data Driven Testing, tests generated by the model are constantly revitalized

Automated MBT

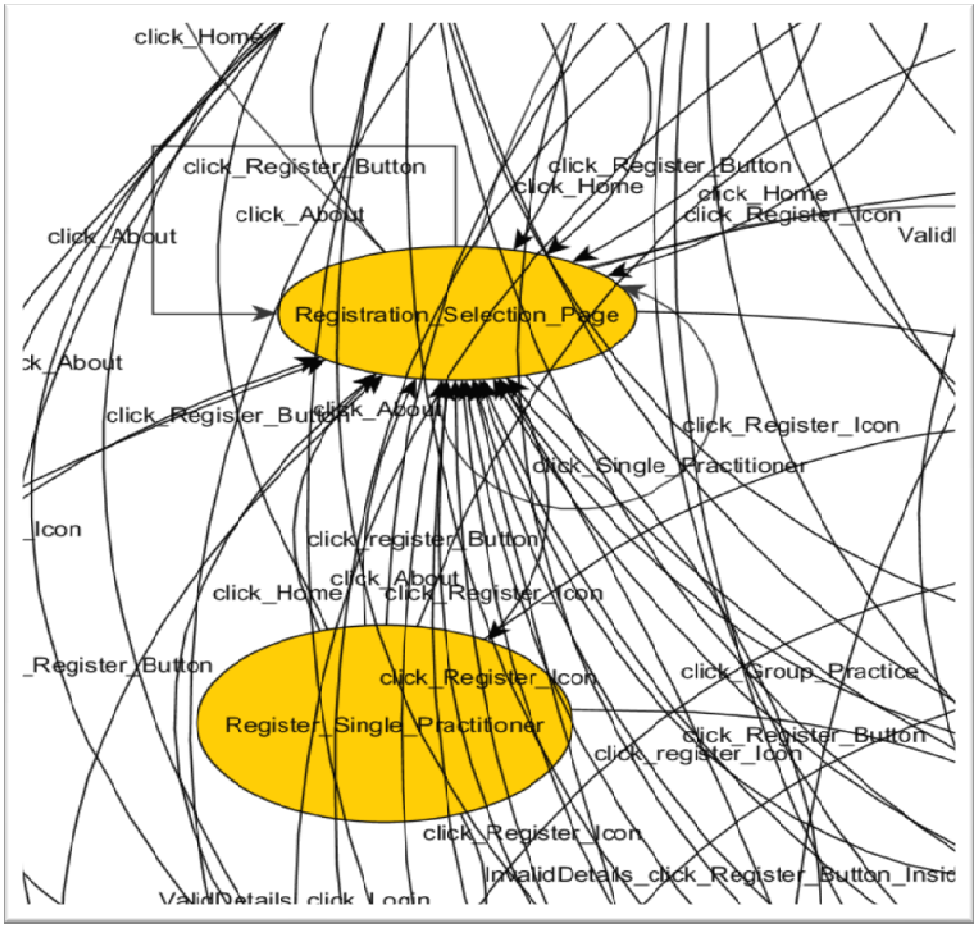
If the complexity of interaction that we have been discussing thus far was limited to trivial examples as discussed then there would be little value in taking this discussion further. However, life is not so kind, the reality is that complexity can be orders of magnitude higher. Consider a realistic, but basic FSM of the actual Web Portal developed by DDI Health website as depicted in Figure 8.

Figure 8 Basic Web Portal FSM



The FSM in Figure 8 is not complete; it is only taking one view of the Web Portal – yet for the exercise consider the number of potential paths. The complexity of behavior is easy to see in Figure 9.

Figure 9 Closer Up View Web Portal



FSM

Again this is still a trivial view of the website in as much as it focuses on only the reachability of pages one to the other, individual flash presentations, page attributes. There is much more complexity that can be looked at. If you feel that you can adequately test all this wealth of interaction to any great level of coverage manually or with test scripts I would suggest you need to review your expectations.

There are several MBT automation frameworks available. Some are Open Source, others are commercial tools. To name just a few there is: NModel[5], an Open Source C# .Net modeling environment; Conformiq Test Generator™[6] a commercial tool that employs UML state charts to constitute a high-level graphical test script; TestOptimal™[7] a commercial tool using the FSM modeling approach with both a community and a full version license available.

At DDI Health we have successfully applied the TestOptimal™ tool to the verification of a major Pathology Web Portal developed for a major Pathology group in Australia. Through the application of MBT we have achieved automated progression testing, which is a major achievement and at least in my experience not heard of in industry, it is a break from and a significant advancement on traditional automated testing. Traditionally, automation is strictly applied in regression testing. Added to the automation during progression testing at DDI we have also achieved automated regression testing and model based load testing. As you can see the potential for Model Based testing is quite significant and

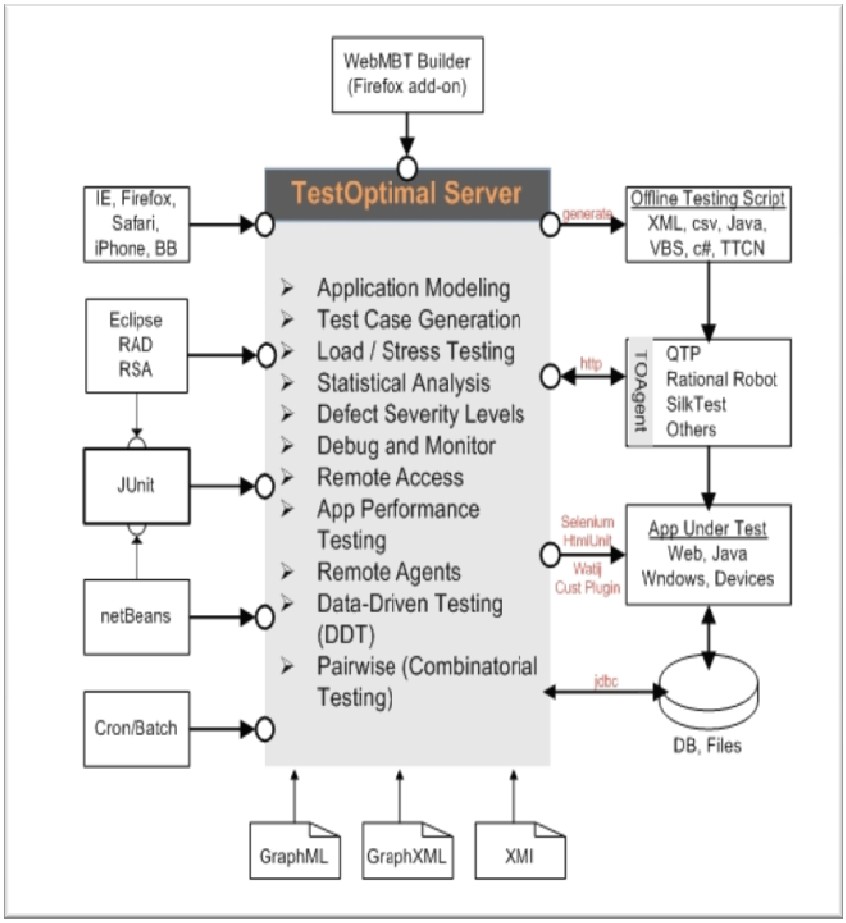
applies to the three pain points for any test organization – progression, regression and load testing, topics that I will discuss in the sections that follow.

TestOptimal™

The TestOptimal™ tool, hereafter referred to as TO for brevity, features a graphic user interface and Java or xml based scripting to embellish the FSM model with the behavior and verification methods necessary to interact with and test a Web Application. Interaction with the Web Application is provided for through a Selenium server that is provided as part of the TO installation. TO further provides for automatic test sequence generation in On-The-Fly and Off-line modes with advanced reporting of coverage, evaluation and performance stats coupled with statistical analysis and capability to support load and stress testing. The architecture provided by TO is as depicted in Figure 10. You may read further on TO by visiting their website www.testoptimmal.com.

TO itself is a Web Application, and is compatible with most web browsers although more commonly used through Internet Explorer or Firefox.

Figure 10 TestOptimal Architecture



Visit http://testoptimal.com/TestOptimalArchitecture.html for a larger image of Figure 10.

Model Based Test Considerations

Model based testing as an approach requires some thought into who will undertake the modeling activities and how the models should be generated. Model based testing requires the attention of very enthusiastic testers. Generating models that are complete and highly capable to addressing requirements verification is a complex task. Further, it should be realized that generating models is in fact a development activity the purpose of which is to test.

The requirements for modeling dictate that the staff assigned to this duty should be:

1. Competent testers – this is essential as the testing mindset is crucial in considering all the implications that need to be addressed in respect of the requirements for the Web Application. I do not advocate developers taking on this role
2. Versed in formal test techniques to some degree – at least the concept and application of Finite State Machine for the purpose of test derivation should not be a foreign concept
3. Capable of designing
4. Capable of writing code

The need for management support cannot be overstated, as with anything new the importance of the new approach needs to be under lined by strong management appreciation. Testing has to be seen as an important and integral task central to achieving a quality product.

To support MBT you need to ensure that the modeling team is involved with the application from the inception or requirement elicitation stage of the development lifecycle. This is crucial as you will have key needs that need to be captured along with every other stakeholder's needs. Key factors that must be heeded and delivered are:

1. Element IDs should be **explicitly** created by the application or by a human being.
2. There should be **no reliance on a UI framework** to create element ids. No non-specific element id's, for example id_255. This sort of non-descriptive element ID tends to cause two problems:
    a. With non-explicit element id's each time the application is deployed, different element ids could be generated and the result of this is that any code built within the model that references the id will likely fail between test runs.
    b. Non-specific element id's make it difficult to keep track of and determine which element ids are required or used within the model
3. GUID's must be explicitly created and not dynamically created on each reload of a page

Treat the models you generate, their code and any associated datasets just as you would treat any other software development artifacts – put them under source control. Regularly check-in and check-out your models to ensure you do not suffer the catastrophic consequences of losing a day or more of work due to hardware or other failure.

Trace your requirements to the models that address them. Make sure you apply good requirements traceability.

Start small. Don't bite off more than you can reasonably handle, gain confidence with smaller tighter models, these also demonstrate competence and garners more management support. Look for the lowest hanging fruit first when modeling. You can always add more complexity as your models evolve.

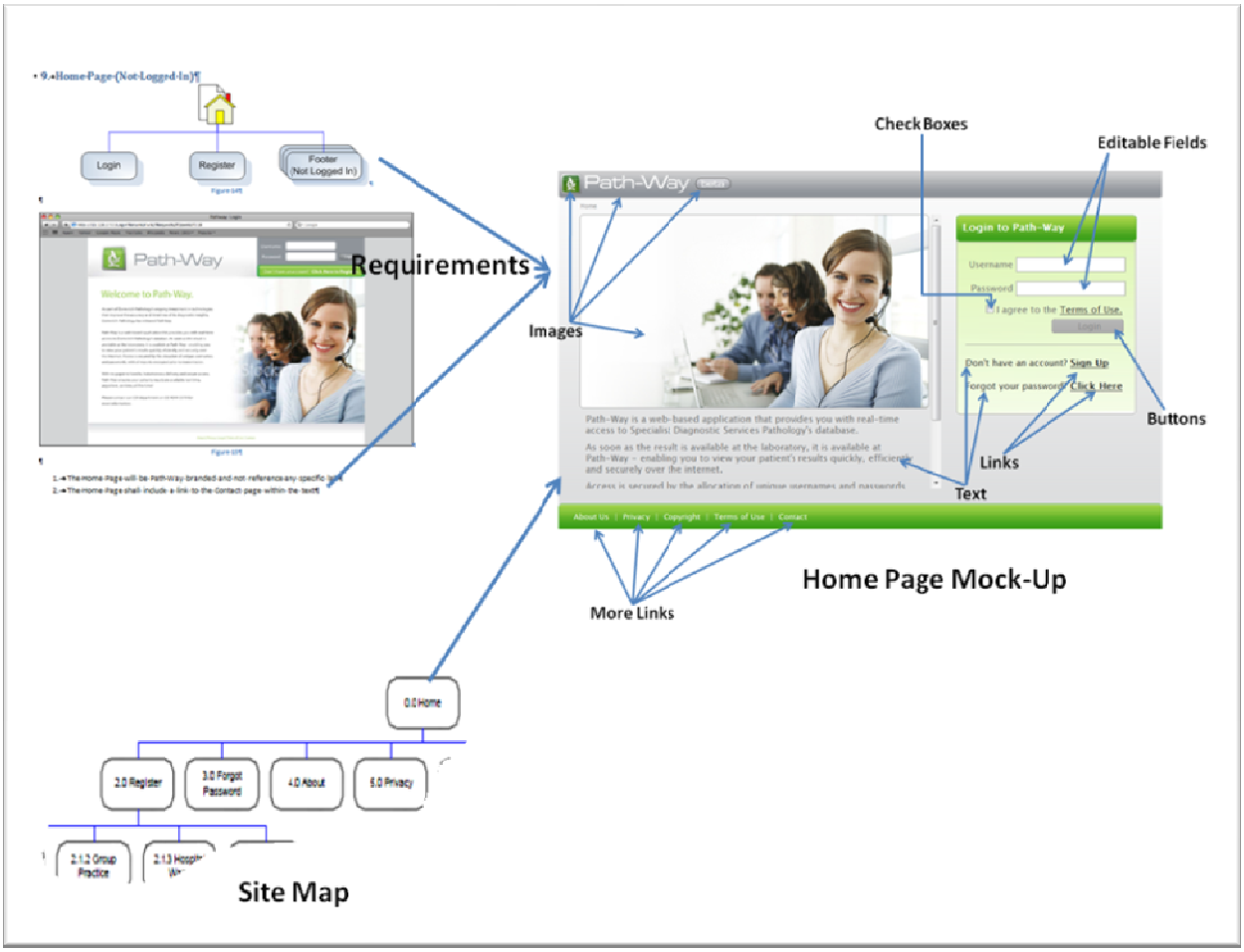
When starting off make sure you have test coverage for the same areas you are modeling via alternate strategies to ensure that you do not leave yourself exposed.

A pilot project is the ideal way to get into MBT or failing that a localized feature set. The measures discussed here are the basic entry level elements to successful MBT.

How To Model

Start with the requirements (if you have them), review the site map for the web application if one exists and look at all mock-ups and wire frames provided. Avoid at all cost modeling based on the actual implemented application. Do not model the application behavior based on the behavior you observe in the application itself. This can be an easy trap to fall into and you need to ensure that you are wise to this. Break down the web application based on the individual pages. Consider Figure 11, the inter-relationship of site map, requirements and mock ups come together to provide a full picture of the scope of the model.

Figure 11 Start To Modeling



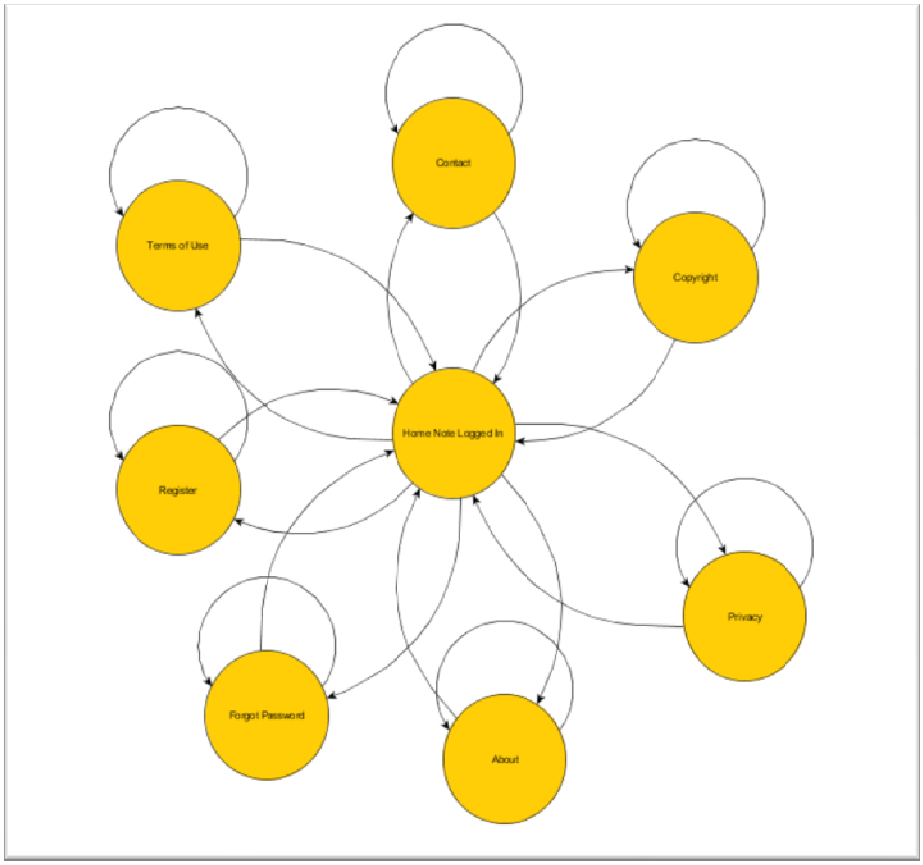
Take each page and generate an FSM model with the page of interest being the starting or initial State. Depending on the page, potentially it may also be the ending or final State. Develop the FSM such that it encompasses the Starting State (page of primary interest) and extend the transitions to all other States (pages) which can be reached directly from your State of primary interest to exactly one level higher and one level lower as depicted on the site map.

In the example provided in Figure 11 the full site map would indicate that the Home page at level 0.0 can directly reach:

1. 1.0 Terms of Use
2. 2.0 Register
3. 3.0 Forgot Password
4. 4.0 About
5. Privacy
6. 6.0 Copyright
7. 7.0 Contact

The resultant FSM model could therefore look like Figure 12Figure 12 Finite State Machine 1.

Figure 12 Finite State Machine



1

Your interest is to fully verify the requirements of your primary page (State) of interest and to confirm that off page visits to neighboring pages (States) are achievable – ordinarily that is as far as your interest or time goes into modeling the Transitions to and about States of neighboring pages. However, if the requirements reflect a deeper association between the pages or highlight an inter-related functionality or dependency between the actions taken on the separate pages then your modeling must take this into consideration and verify this inter-relationship as fully as possible.

This results in a series of slightly overlapping FSM models but ensures that no major or trivial association is missed. This provides for a systematic approach and ensures close collaboration between the separate individuals providing the modeling.

Employing A Modeling Tool

It is all well and good to talk of modeling as discussed above. You can easily do your modeling using pencil and paper, but that will miss the objective which is to get the model to automate your testing. The tool used by DDI Health to achieve this is TO. How to get the model into TO? There are two ways to achieve this:

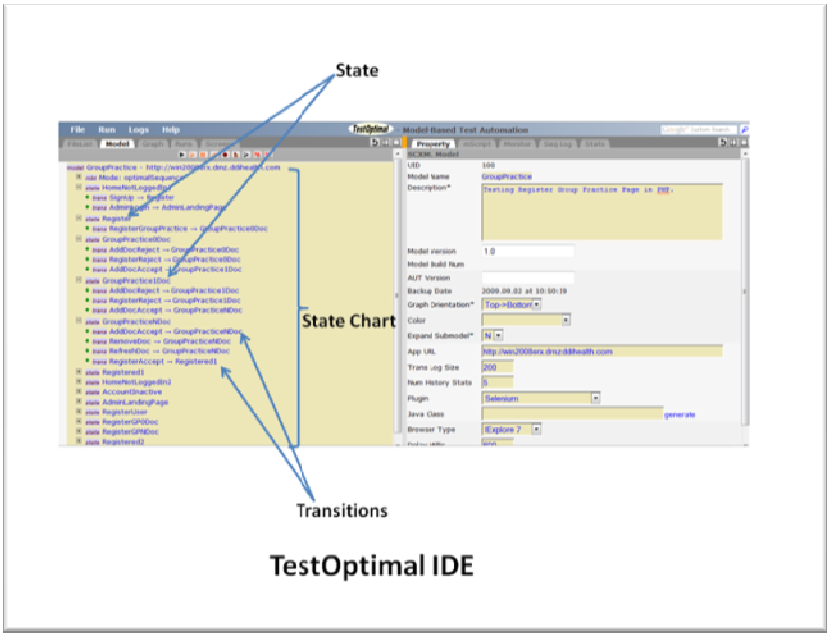
1. Generate a model of the behavior in a FSM format that can be imported by TO

2. Create the FSM model directly from within TO's resources

TO is capable of importing GraphML and GraphXML formats. There are very many licensed and freely downloadable graph editing tools available that are capable of producing graphical models which may be exported as either GraphML or GraphXML formats amongst others. The external graph editing tool used by DDI Health is yEd[9] from yWorks. In fact the examples of FSM provided throughout this paper have been produced in yEd.

The alternate and possibly better approach is to generate the FSM directly from within TO. The approach taken by TO is to represent the FSM in a state chart format

Figure 13 TO Interface



Whether imported from an external 3rd party graph editor or directly created within TO the end result is an FSM in a machine readable format. However, that is not the end of the story there is still work to do – no one said you were going to get away without writing code. The graph editor or the state chart editor of TO, only produced an abstract skeleton or logical framework representation of the potential behavior, but there is nothing in the model so far that provides for any concrete interaction with the real world. That task will always require coding. Fortunately, however, once the abstract model is entered into TO, TO produces a skeleton XML file that is ready for addition of your code utilizing TO's methods to achieve the outcomes you intend to accomplish. TO employs what its creators term mScripts, a scripting language based in XML, but extended with their own methods. You can review these methods by visiting TO's mScript[10] Function help page. An example of mScript from TO's mScript editor is shown in Figure 1

Figure 14
mScript

The example shown in Figure 14 shows the style of mScript that we have adopted at DDI Health. In order to ensure the maintainability of the models as well as future re-utilization, the decision was made not to use any hard coding within the models. That is to say that any parameters that are derived from sources not directly controlled within the model (such as element id's, GUID's, xpaths etc.), or functions that are called repeatedly or are subject to change, were parameterized.

Consider line 276 in the example provided, Figure 14. The mScript method for "type" or enter a value into an editable field is explicitly:

type (java.lang.String locator_p, java.lang.String keyString_p)

We could have put explicit values for the *java.lang.String locator_p*, which is the element locator and an explicit value for *java.lang.String keyString_p. That would result in an action mScript like:*

<action code=$type('xpath=.//[@id='usename']','Joe123bad')>*

Where *.//*[@id='usename']* is the element locator for the username field and *Joe123bad* is the explicit user name. However this style of coding is seen by us as very poor practice and prone to being easily broken, especially when developers change their code and element locators change commensurately.

So don't do it is the simple advice, keep your coding as dynamic and manageable as you can. For this and another very important reason, we wished to drive our models with multiple datasets so we can take advantage of the Data Driven approach to testing, we decided on paramterizing out anything we wished to manage and control external to the model.

So to achieve our aims the mScript we generated in line 276 is:

<action lid="276" code="$type('$getData('ElementNamesDS','UsernameEdit')','$getData('BadDS','Username')')"/>

In this example *$getData('ElementNamesDS','UsernameEdit')'* fetches the the xpath locator for the editable field (username) and *$getData('BadDS','Username')* fetches absolute value that needs to be entered into the field.

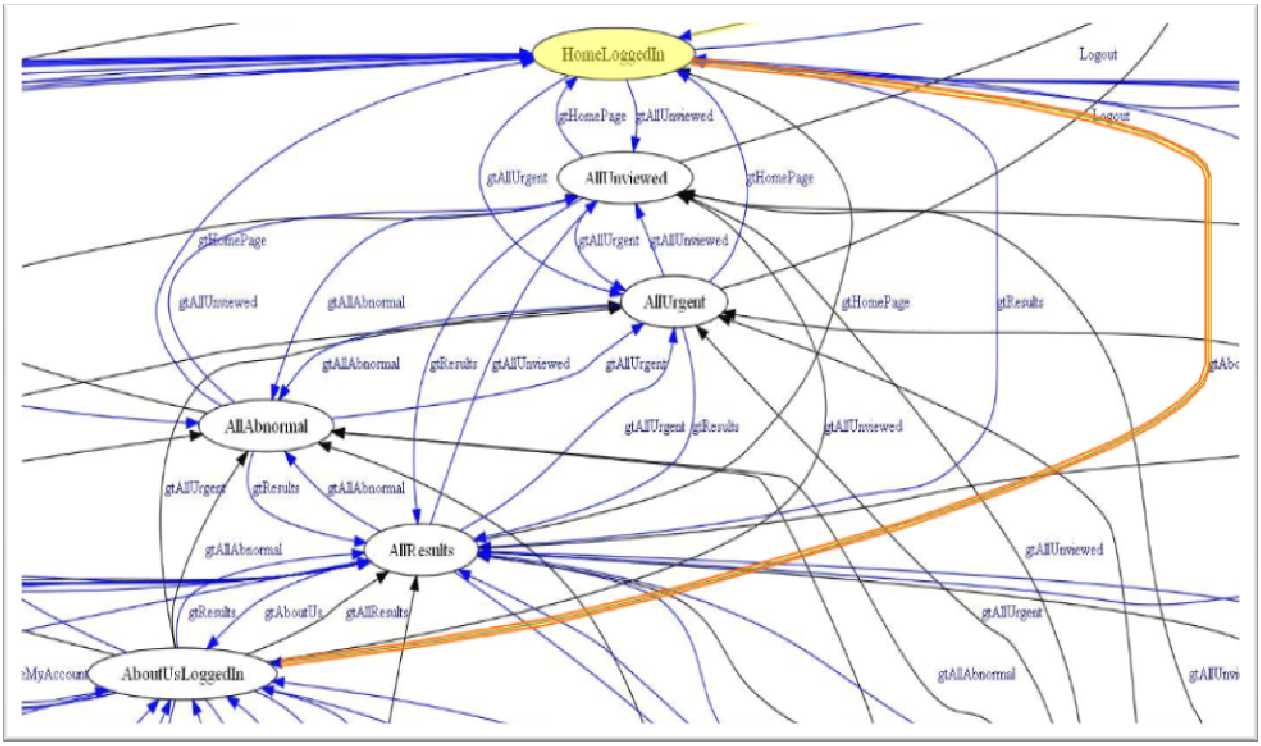
An important feature of TO that, at DDI Health, we have found most helpful is the Model Graph which TO generates based on the model that you have defined. This graph has two unique properties that are essential in any modeling:

1. Auto-navigation based on a preset delay between Transitions. This is extremely useful in:
    a. Communicating with management and other stakeholders
    b. Training and educating testers
2. Debugging and validating the model. By stepping through the model and observing the State transitions and the actual Transitions firing, whilst simultaneously observing the AUT under model control it is relatively easy to determine whether the behavior observed is consistent with the specifications and requirements and whether model and AUT are in synch.

The graph of the model highlights the Transitions as they are fired and also highlights the States as they are occupied. This makes navigating the model very simple and relatively easy to compare against the AUT in operation.

An example of the Model Graph feature of TO is provided in Figure 15.

Figure 15 TO FSM Graph In Action



The Abstract Model

The points raised above should be heeded by anyone attempting programming for modeling. **The additional benefits to this approach are that models may be built in advance of the actual implementation using the requirements alone.** It is understood that in the end concrete values must be supplied to the model. These concrete values of element ids, GUID ids, Xpath's can all be derived from the actual AUT when it is available. If these were available prior to the AUT being produced then these could have been utilized earlier, reality of course is that not even the developer's know these ahead of time. So concretization can in general only occur once a build is released to QA. However, It is unlikely that during progression testing that these automation steps can be adequately accommodated. not the models can still be built to a high degree of completion without these due to the parameterization discussed already.

Concretizing The Model

Once you have an abstract model in TO you need to concretize it. This you can do surprisingly easily and efficiently. You have two paths to follow in this regard. You can either use MS Internet Explorer or Mozilla Firefox. Both these browsers have developer tools either as add-ons or integrated natively. Our preference at DDI Health has been to use Firefox which has an exceptional wealth of developer aids that just make the task of concretization so easy. It has to be said that all Internet Explorer has to offer in the form of a developer aid is the Developer Toolkit which can be downloaded for IE7 or is native to IE8 and this developer tool kit it has to be said, comes in as a very poor second to the add-ons

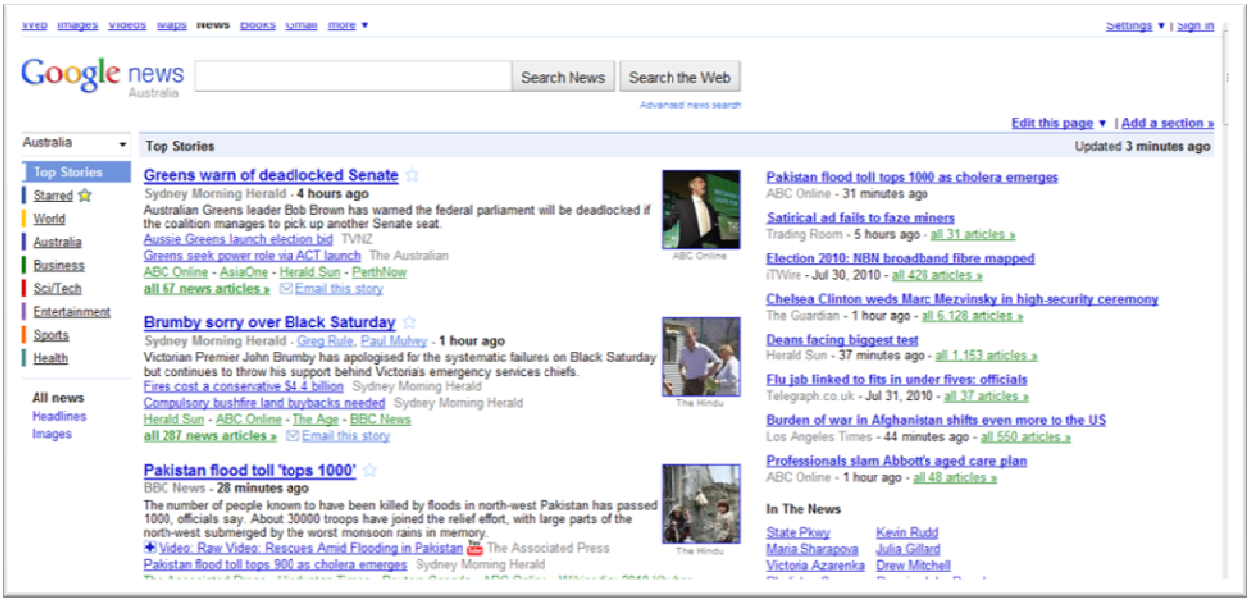
available to Firefox. Certainly we employ Internet Explorer as we do have products at DDI Health which are exclusively hosted through Internet Explorer but it simply is not as easy to work with.

In respect of Firefox there are some key add-ons that you must ensure that you have, these are:

1. DOM Inspector
2. Firebug
3. FireXPATH – an extension to Firebug
4. XPATH Checker
5. XPather

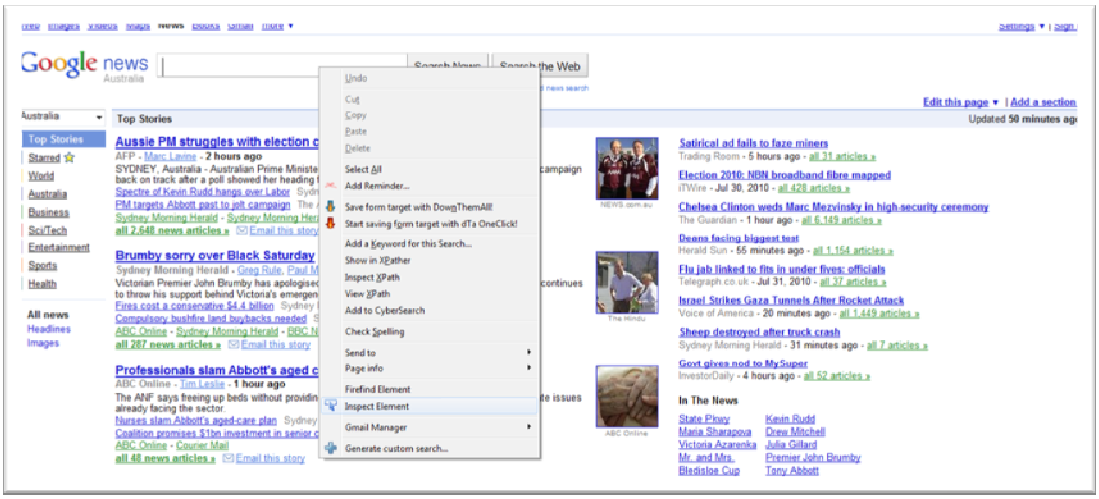
These add-ons are free and absolutely crucial to your activities here after, reference to Firefox will be exclusive from this point. So how do you actually then get the concrete values that you need? Start by starting your application in Firefox. For the sake of this exercise we will look at Google News, Figure 16.

Figure 16 Google News Web Site



Next you want to extract the concrete value for the Search News Field. It is simply a matter of right-clicking and selecting Inspect Element from the drop down list.

Figure 17 Inspect Element



Once Select Element is chosen Firefox opens the Firebug tool. Note that the Xpath tab is selected, if it is not then select it. Select the selection icon and focus on the Search News field. The Xpath is instantly displayed.

Figure 18 Concrete Xpath

The Xpath in this instance is .//*[@id='page-header']/table/tbody/tr/td[2]/form/div/input[6]. Now as well as Xpath's, you can utilize the full features of Firebug and explore the web page code and extract all attributes, image references and so on. The process is as simple as that.

In this manner you can systematically work through and very rapidly concretize your models. Of course the process of concretization does take time, but remember your abstract model and code are, or

should be complete. All you are now doing is providing the references to the concrete element values. The time required generally is under a day. Then you can execute your model.

Model Execution

Executing the model through TO is a relatively simple process and depending on the complexity and extensiveness of your model it will execute mostly as fast as you set it to run. The models may employ any one of several graph traversal algorithms to generate the test sequences on-the-fly. These are:

1. Optimal Sequencer - uses mathematical optimization algorithms to generate test sequences that cover all of the transitions on the model with the least number of steps.
2. Random Sequencer - generates test sequences by randomly choosing a transition from each state to get to the next state. This process is repeated until any one of several stop conditions evaluates to true.
3. Greedy Sequencer - similar to Random Sequencer except that it prefers un-tested transition over already traversed transitions.
4. mCase Optimal Sequencer - uses mathematical optimization algorithms to generate the test **sequences** to cover 100% of the transitions included in an [mCases | mCaseNode]. The test steps defined in each mCase need not be consecutive or in any order, the system will find the optimal way to fill the gap with transitions from the model. The algorithm used is an extended version of Chinese Postman Walk algorithm to handle optional transitions and multiple traversals.
5. mCase Serial Sequencer – mCase Serial is similar to mCase Optimal although the order of the transition in mCase are explicitly executed in the order they are defined

Model execution may further be set to be Transition based or Path based.

6. Transition based - generates sequences to coverage of all transitions in the model or in mCase at least once dependent on the sequencer selected.
7. Path based (Paths) - generates sequences to not only cover all transitions at least once, but includes permutations between incoming transitions and outgoing transitions. This type of coverage is much more extensive.

The model execution discussed thus far provides for highly capable and extensive functional testing within a rapid development environment.

However, a further benefit of model based testing and especially from within the TO tool is that each and every model can be repurposed to another area of need; load testing. Unlike other load testing techniques, which rely on capturing interaction sequences from a user to an application, and then replaying these sequences or traces as many times over as required to produce a load consistent with a user base of N-users. TO can be used to launch each model on as many threads as you require (this requires the TO runtime server license and purchase of the number of threads you need to replicate). Each model represents a very realistic facsimile of an actual user interacting with the application and in as much as you apply Data Driven techniques as well you can arrange for a very massive, very realistic load on the application. So your load testing comes almost for free, once you have built your models.

The end result of the execution is a series of files. Depending on your code you may have generated a lot of Log Message which will be found in the mScript Log file, apart from this other files generated include Selenium Server and Console Log Files all of which are instructive and useful in debugging

failing models. Amongst the most useful log files produced are from the Stats Tab of the TO application. This tab retains the full execution results that indicate useful metrics such as page average load times, number of passed and failed requirements, which is crucial for any professional test organization and is essential for any formal reporting.

Further Benefits of Model Based Testing

Defects removed in stages prior to implementation are 100 to 1000 times cheaper than discovering and repairing defects post implementation. So when you consider that:

- Requirements gathering, analysis and architectural design accounts for between **60%** and **70%** of all defects introduced into a software product (from studies conducted by Kirova[11])
- Coding accounts for **30%** to **40%** of defects discovered in software products (Kirova[a][11])
- Up to **80%** of all software development time is spent on locating and correcting defects (includes test)(NIST 2002)

Any means that can improve the situation in respect of reducing seeded defects and removing defects with in specifications is of great importance.

Beyond the significant benefits gained through the resultant automation arising from the modeling and Model Based Testing there is another important and sometimes overlooked benefit. Modeling has the distinct capability to reveal defects in requirements and specifications. These defects include deficiencies, internal and external inconsistencies, ambiguities, omissions, misstatements, contradictions and a whole host of other "sins" that all requirements are prone to experience. The benefit to the project, not just the test organization, is that very early in the development lifecycle, and potentially well in advance of the development team, the models can and do, highlight problems in the requirements.

This is not surprising, on a one for one comparison with Fagan review process and modeling, in an experiment, 100 requirements defects were purposely introduced into a functional requirement document (inaccuracies, ambiguities, inconsistencies, incompleteness, etc.)

- Fagan[12][13] review process detected 34 defects (consistently), that is an efficiency of **34%**
- Model approach detected Fagan 34 defects **PLUS** 56 other defects, that is 90 defects and they were discovered in the first pass, that is an efficiency of **90%**

The latter results speak for themselves. Robust static Reviews such as Fagan reviews are at best 34% efficient and are capable of detecting only, (and notably not every instance of)

- Requirement ambiguity
- Requirement incompleteness
- Requirement Inconsistency

Models on the other hand as opposed to static review methods have been shown to detect, consistently:

- Requirement ambiguity
- Requirement incompleteness
- Requirement Inconsistency

Plus,

- Feature/Function Interaction
    - Inconsistency
    - Error
    - Deficiency

Conclusion

At DDI Health we have very successfully and competently adopted Model Based Testing as a systematic approach to providing us with:

1. Highly adaptable test generation capabilities
2. Very rapid model updates and test regeneration and execution
3. Highly effective functional testing
4. Automated progression testing – this on its own is highly noteworthy
5. 100% regression testing (as far as modeled behavior goes and that at this writing for the Pathology Web Portal exceeds 95% of all requirements)
6. Extremely short test cycles in the order of a day for exceptional coverage
7. Repurposed functional tests that acted as highly capable load tests almost immediately
8. Happier testers
9. Secure and satisfied test managers
10. Satisfied stakeholders

The days of working late nights and weekends seem to be fading for the projects that are Model Based Testing focused. We plan to further develop our Model Based Testing capabilities and currently have seven test engineers who are trained or being trained in this technique.

We certainly rely on TO, however we do have plans to introduce other MBT technologies. We believe that solid knowledge in MBT needs to be backed by multiple tools and techniques to allow us to be adaptable and not allow the weakness of any one approach limit our capabilities.

We count the MBT approach applied to the Pathology Web Portal through TO have been a salient success story and it is our view that we shall see many more such successes.

Do very seriously consider MBT for all web based applications and think also of non-web applications as well. Model Based Testing does work, has worked for us, and does provide serious efficiencies. TO is a capable tool that you should investigate, we have been impressed with the quality of the support we have received from the TO support team. The responsiveness from the support team at TO has been exceptionally timely, professional, courteous and very helpful.

Good luck in your endeavors.

References

[1]. Software Engineering Book of Knowledge IEEE – 2004 Version

[2]. Dan Mosley, Test -Rx Standardized Software Testing Process Methodology, CSST Technologies, (Mosley, 1997)

[3]. NIST online Finite State Machine definition

[4]. DDI Health website: http://www.ddihealth.com

[5]. NModel online: http://nmodel.codeplex.com/

[6]. Conformiq Test Generator™: http:/www.conformiq.com

[7]. TestOptimal : http://testoptimal.com

[8]. Beizer, B. (1990) Software Testing Techniques, 2nd Edition

[9]. yEd: http://www.yworks.com/en/products_yed_about.html

[10]. TestOptimal mScript Function Help Page: http://testoptimal.com/wiki/index.php?mScriptFunc

[11]. Vassilka Kirova, Ph.D: http://web.njit.edu/~kirova/

[12]. Michael Fagan Associates: http://en.wikipedia.org/wiki/Fagan_inspection

[13]. The Defect Free Process: http://www.mfagan.com/process_frame.html