CONFORMIQ

Automating the Automation

The Bizarre Truth!

Complicated & Confusing taxonomy of Model Based Testing approach

A CONFORMIQ WHITEPAPER

By Kimmo Nupponen

TABLE OF CONTENTS

The Context

Introduction:

To most engineers the term MBT, for Model Based Testing, usually means using graphical models to be the basis for test generation. While this doesn't hold true for all the different tools, for those that it does, the usability, capability, and benefits vary widely between them. Many tools model the test flows or even the test cases themselves by having the user think of the application flows.

Once the flows are drawn from the requirements, test steps and validations are manually added. These approaches deliver some value but often take more time than just writing test cases manually. The real trick is to have the MBT software automate the thinking of the test case design and then automatically generate the test cases, test steps, and validations, without any user involvement, for direct automated test execution.

Unless the MBT application itself thinks of the test information required for automated execution the efficiency of the testing process may not be great enough to motivate companies to make the digital testing transformation. Further, not achieving sufficient gains with one tool shouldn't mean companies stop looking for a better tool. They just need to do their homework because these tools are very different "under the hood".

The Approach

Know the difference between different MBT tools:

Model Based Testing is a quite confusing term and can be interpreted in multiple different ways where one interpretation is as accurate as the next. Model Based Testing, or MBT for short, is actually a term that captures multiple vastly different approaches under one big umbrella. Over the years, we have been educating the industry about these approaches, their weaknesses and strengths, and we have been using taxonomy in order to classify different approaches based on their core capabilities.

According to this taxonomy, the three main MBT approaches are –

- Graphical test modeling approach
- Environment model driven test generation
- System model driven test generation

There are also others but these three are the main approaches.

This taxonomy unfortunately is quite incomplete and problematic at best. Firstly, there are approaches that do not fit into these buckets in any practical way while still being "model based". As an example, we can take combinatorial testing tools that create test cases using a model as inputs. Here, the model is something that does not fall into any of these buckets as the model actually captures data attributes, their respected values, some restrictions, and so on. Additionally, even within a certain bucket there are huge differences between approaches and tooling.

While choosing the best approach, always consider the solution which fits tightly under the “system-model driven” approach. During the past few years, a lot of vendors and solutions have emerged that claim to be “system model driven” and automate test design while actually being extremely limited in their core capabilities. Just by looking into classification, one cannot really tell what a particular approach is good for and where it lacks capabilities.

Indeed, it can be very difficult to see differences between tools unless you look a bit deeper and ask the proper questions from the vendors. This whitepaper aims to highlight some of these main differences and questions that everyone interested in MBT should raise while evaluating different tools.

The Right one

Things to Consider while choosing the MBT solution:

With system model driven approaches, the practice is to focus on modeling (functional) requirements of the system. There are today a handful of approaches from few vendors where users indeed model the requirements so, in that way, yes they do fall under system-model driven approaches.

The models are graphical, you annotate them with requirements and tooling creates test cases out of them with requirement traceability, sure. Actually all the model based testing approaches can produce the same end result -- that is they can all be used to create executable test cases and test documentation. However, this is not the main point. The key here is what users need to do in order to get those tests out. This is where there are significant differences between approaches even within the system-model driven umbrella. (Technically, the approaches by few vendors are actually based on exactly the same algorithmic and technology foundation as those tools that fall under tester/environment-model driven approaches in the taxonomy described in our * previous white paper. It is just that in their way of approaching the users they focus on functional requirements.)

**Previous Whitepaper is already published and available on our website. Read the paper by clicking HERE.*

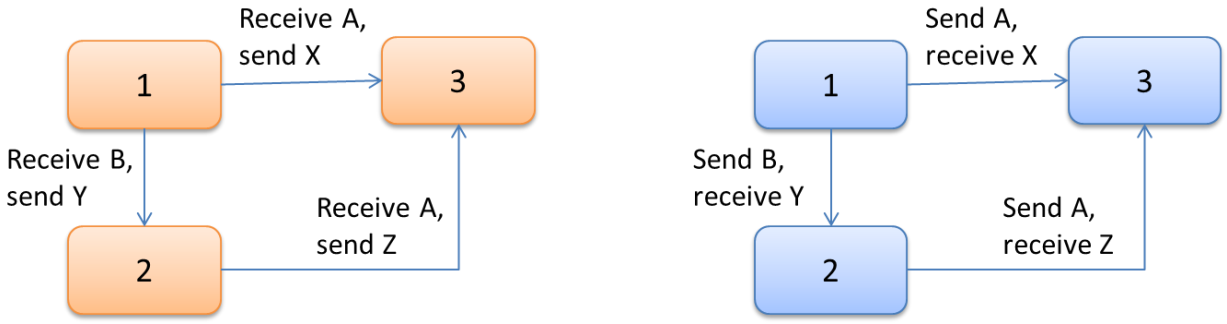

Figure 1: System models

Explanation: System model on the left hand side created in the form of a finite state machine and the same system model repurposed as a tester model on the right hand side. Two things happen here: inputs and outputs are switched, and the state numbers in the tester model now refer to internal states of the system under test to be verified. Path traversal procedures can now be executed for this tester model in order to produce test cases.

This results in the polynomial-time generation of a polynomial-size test suite. As expected, the system model and the tester model are the same model for all practical purposes, and they are both computationally easy to handle. This explains the wide success of finite-state machine approaches for relatively simple test generation problems, and also the confusion that sometimes arises about whether the differences between tester models and system models are real or not. For explicitly represented finite-state models, there are no fundamental differences.

The approach implemented by Conformiq is driven by semantics of the model, which first of all means that the graphical structure of the model is not used to guide the test generation at all but only the logical meaning of the model. This is in stark contrast with simpler state machine / activity diagram driven test generation approaches where the structure of a (typically only one) state machine / activity diagram is used to generate a sequence of tests that correspond to different paths through the graphical structure via straightforward execution of the model. This constitutes tests in their generation process. While on surface this may sound like it's merely a technicality, this is a very important point because only the fully semantics driven approach can tolerate models where the high-level control flow is deeply dependent on data values.

The above reasoning goes directly down not only into test data generation but to the level of test design automation. Automatic test data generation, or lack of it, is the crucial part that often leads to mediocre or poor quality testing and improved efficiency. With a majority of approaches out there the test data generation and test output verification problems are not solved in an efficient manner (i.e., not generated automatically by the tool itself), and generating inputs and verifying outputs ("test oracle") is left completely to the user to manually think of and include in the model. Now with those approaches, the models need to either contain hand-crafted procedures both for input generation as well as for output verification, or then the data design must be done entirely outside the context of modeling, and both are difficult, introduce large risk and suffer from maintenance and modularity and thus reuse problems.

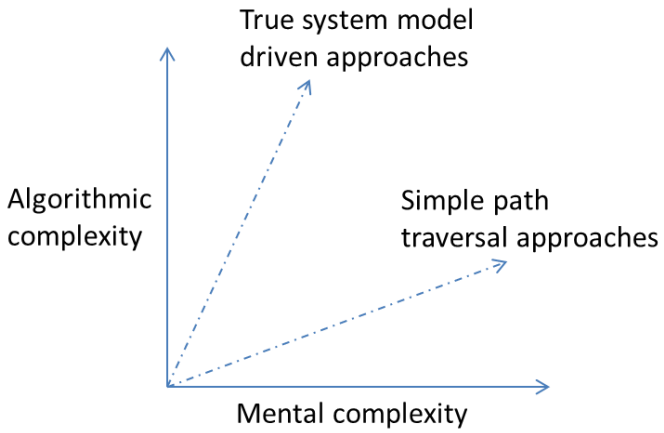

Figure 2: Scalability of the approach

Explanation: For the pure system model driven paradigm, the main scalability problem is algorithmic complexity. The complexity grows when the application to be tested becomes more complex from testing perspective (leftmost dashed arrow). For approaches based on a simple path traversal—whether advertised as system model or tester model driven approach—the main scalability issue is the cognitive difficulty of producing and maintaining good models (rightmost dashed arrow)

Consider how many systems you can think of whose behavior is not integrally coupled with the input that it processes? Actually, most applications and systems have infinitely many different ways to interact with them, at least on a detailed level, and you need to make a (conscious or not) decision on which scenarios to use for testing the system and which not; which data parameters to use on the selected scenarios, the expected outcome, and so on.

A solution that leaves data design as an exercise for the user is bound to be insufficient with severe shortcomings in its capability to produce good quality tests or will take extended time to model. For example, real world applications have conditional branching and iteration in their flows so if explicit data are added to tests generated from the process flow, the first challenge is to "fit" the flow and the data, but especially afterwards when flow changes occur and you need to redesign and "refit" the data. Great manual effort is needed to select the exact always correct data.

The key point is that you cannot think of control flow and data in isolation and therefore separating the process of test design into flow and data design silos is problematic. This once again stems from the simple fact that the control flow of virtually all real world systems is deeply dependent on data values, and we argue that it is extremely difficult to create good quality tests if you do not deeply couple these key aspects in your test design efforts and generate optimized test cases from both combined.

Indeed, the approach chosen by Conformiq allows the tool to consider (through highly complex algorithmic operations) all the logic aspects of your system. It will automatically figure out how the control flow depends on data and vice-versa, and it will automatically produce test cases (and fully executable test scripts) that accurately cover those logical and behavioral details of the system. We do recognize that there is a need for providing a way to "inject" the model with production and provision data so we do provide the means of importing data from external sources, but that comes after verifying that the system logic is thoroughly tested and does not reduce the value of the test generation approach driven purely by the semantics of the model. In simplified terms, Conformiq software understands the logic of the SUT from the model and thinks of the test cases considering data and flow together.

> “Conformiq MBT solution automates the thinking of the test case design and then automatically generates the test cases, test steps, and validations, without any user involvement, for direct automated test execution. Virtually, all other MBT tools available in the market do NOT deliver these capabilities and this critically important distinction is why this paper was written.”

The Crucial element

How is Data Handled?

Another very important point on data is Conformiq's unique use of symbolic data dependencies used to verify and select test data provided by the user and/or compute unspecified data. In practice, this enables much more efficient modeling and model reusability. This is the first requirement for symbolic execution. If you cannot treat the data in a symbolic fashion, you cannot really apply much abstraction in modeling (symbolic meaning that we do not a priori know the concrete value so we must treat it symbolically; for example, an integer that we know nothing about is treated as a symbolic element meaning that it holds any integral value and we only know the actual concrete value later). Having the capability to abstract data in this way is very important. If you cannot abstract data, what you essentially need to do is to enumerate all the possible accepted and unaccepted values explicitly in the model, which makes modeling tedious, error prone, unrepeatable, etc., plus making models hard to understand, reuse, and maintain. Needing explicit data leads to test errors if application loop backs cause the data to change. How (if) we handle iteration in the model is a very common and important question for just this reason.

Consider the following web shop application as a simple example to explain the importance of this capability. In this example the system accepts various items to be placed into and removed from the shopping basket in any order. Using Conformiq's Designer or Creator, the application is modeled just like the real system would operate without "hard wired" data. The user can checkout or cancel, etc. in any order with any items. The tool understands the semantics. With a less capable approach, users must enumerate all the paths; i.e., put this particular item into the basket and checkout; put in this particular item and remove it and then checkout, etc. Every step with every data element to be tested must be explicitly modeled. You can see that in a real world application with comprehensive testing the modeling difference and reusability would be quite different between these approaches.

The USP

What makes the Underlying Test Design Engine Different?

It's not just test optimization from reducing the number of test cases. This is simple and most MBT tools perform this task. It is actual test design. The real capability differences come from deeper within the engine. Conformiq's Designer and Creator are unique because they are based on a custom crafted semantics driven, symbolic state space exploration algorithm. This is the only known solution that robustly generates both test inputs and outputs from a system model without user intervention.

Symbolic data dependencies are handled by constraint solving which is used to verify and select test data provided by the user and/or compute unspecified data. Test generation is guided by a deep state space analysis of the behavior implied by the model. Controls are embedded within the tool to limit the problem of state space explosion. Test selection is based on model driven coverage criteria and combinatorial optimization from the explored part of the state space by the tool automatically selecting paths that lead to testing goals. Thus there are major Conformiq engine differences delivering large benefits in real world use as compared to the simple graph walker (i.e. test flow) tools previously mentioned.

In essence

Summary:

Selecting Model Based Testing products on their modeling paradigm is a start but is just the tip of the iceberg, so to speak. These tools do not exist in isolation so much more needs to be researched on how the tools solve your problems. Classifying model based testing tool approaches only based on high level characteristics is simply not sufficient to answer those fundamental questions that we brought up earlier in this discussion. A high level comparison misses critically important engine differences.

In the end, it all really boils down to what the user needs to do in order to get tests out. Does better coverage really matter? Does efficiency really matter? We think they should and this is probably the true difference in Conformiq's approach.

To stay updated, follow us

About the Author:

Kimmo is the leader of Conformiq's R&D team and has responsibility for Conformiq's R&D operations. For more than a decade, he has been the chief architect, inventor, and developer of the Conformiq's test generation technology used by both Conformiq Creator and Designer. He is also the product owner of Conformiq Designer. Kimmo holds a M.Sc. Tech. degree from Helsinki University of Technology where he studied theoretical computer science and software systems.

www.conformiq.com