What is Important When Selecting an MBT Tool?

Interest towards model-based testing has increased quite significantly over the years as people have started to reach limits of traditional approaches and at the same time started to see and understand the benefits that applying MBT can have to the quality assurance function.

The first and maybe even quite surprising observation that people often make when they start to look at MBT is the variety of completely different approaches and the vastness of both academic and commercial tools. MBT actually means numerous different things and approaches. In a loose term, model based testing is anything that is based on computer readable models that describe some aspects of the system to be tested in such a format and accuracy that it enables either completely or semi-automatic generation of test cases.

The three main approaches to model based testing are the **graphical test modeling** approach, **environment model driven test generation**, and **system model driven test generation**. There are also others but these three are the main approaches.

All the model based testing approaches above can produce the same end result – that is they can all be used to generate executable test cases and test documentation. However, this is not the main point here. The key here is *what the users need to do in order to get those tests out*.

Graphical test modeling is simplest of the approaches listed above and is actually nothing more than *modeling the test cases themselves in a graphical notation.* **Environment, use case,** or **usage models** describe the *expected environment of the SUT*. That is, these models describe how the system under test is used and how the environment around the system operates. These models represent the tester – not the system that we are testing. The models include *testing strategies*, that is the input selection, and hand crafted output validators, or *test oracles*. The third main approach to model based testing is **called system model driven test generation**. Here the idea is that the *model represents the actual, desired behavior of the system itself*. Conformiq Creator™ and Designer™ are examples of a system model driven approach.

With graphical and environment MBT approaches, the process of test design, that is the process of deciding how to test, what to test and what not, is a manual activity. These approaches to MBT rely on manual test design so they speed up parts of the test design process but still leave a lot of work to the manual process of thinking through all the necessary test steps and combinations, which introduces a lot of risks, such as missed test coverage, and it takes a lot of time, especially when the requirements change. As the intent of this discussion is to expose the reader to the most advanced of these MBT technologies, we will limit this discussion to system model driven approaches since they are the only approaches that are used to actually **automate the test design**. Because there are different automated test design tools, each with sometimes subtle differences from the others, it is useful to understand what is important to know when selecting an MBT tool for automating the test design.

Modeling / tool ease of use

Since all MBT tools and methods start with a model, obviously the modeling notation and environment needs to be such that the end user can understand and feel comfortable working with it. Great test generation features of a MBT tool are close to useless if you cannot understand how to use them. There are many drawing and modeling tools, so it is important to select a tool that "fits your need". Overkill with many non-modeling functions makes learning and using that type of tool more complicated than needed and drawing tools that don't restrict the models to those constructs used for test generation, allow users to introduce non-functional notations that will need to be changed later during the import into the test design tool. The optimal solution is a modeling tool tailored to the user's needs. In this case – system design for test generation. Also learn if a third party modeling tool is required or one comes with the test design engine. Even modest tool costs add up for larger volumes but, more importantly, you will need to continually match differing releases for compatibility and get in the middle of two vendors to get an immediate fix when a defect is found. Is the third party tool worth the risk and effort is a question to ask.

Modeling expressivity

The end user needs to be able to rigorously express the system behavior; an MBT tool that does not allow you to do that is quite useless. For example, does the tool support multithreaded / multicomponent modeling?; Hierarchical decomposition?; Concepts for model reuse?; Advanced arithmetic? If these system operations can't be expressed in the models, the test design tool can't cover them. Ask what are the constructs your SUT needs modeled to express its behavior.

There needs to be a careful balance between expressivity and ease of use since as you simplify the modeling notation for making it easier to understand and work with, you are inevitable sacrificing expressivity and the other way around. This is the reason why Conformiq offers two alternative notations. The more traditional modeling notation which is based on Java and UML is highly expressive and can be used to meaningfully describe extremely complex systems, but requires programming skills, so users need to be relatively technical. More higher level / system level models that do not require such a support for expressivity, can be modeled using simple non-programming Conformiq Creator notation, which then can be used by testers and Subject Matter Experts (SMEs) with little less technical background. Both modeling tools are designed to create system representations for testing without carrying the overhead of additional non-modeling complexity.

Generation of great quality tests

The quality of the test cases is by far the most important thing in quality assurance. If the quality of your tests is low, it really does not matter how fancy your testing processes are or how cool are the tools you are using for test execution. When you are about to automate the test design, you really need to assess the quality of the test cases that the MBT tool produces. One naturally needs to remember here that the quality of the output that the MBT tool generates (tests) cannot be on a higher level of quality than the input (model). Therefore you must also pay close attention to the quality and completeness of the model.

Look for a tool that offers a great variety of different test generation heuristics. Relate these heuristics to your particular needs. If your system is manipulating string values, the tool should have some extensive support for expressing string patterns (such as regular expressions). If it is conducting a lot of numeric calculations, make sure that the tool has support for boundary value analysis and other equivalence class partition methods. The more test design heuristics the tool supports, the more breath of test coverage and flexibility it delivers for testing different types of designs. What do and will you need for full coverage?

Scalability

Test generation is computationally very intensive task. The more complex the system and model and test heuristics used, the more computing resources needed. Then add a very key capability for the tool to automatically optimize the minimum number of test cases from all possible, and the computing resources needed are even higher. This is key, since there may be thousands of test combinations generated based on your test coverage heuristics, but you should want the tool to determine the minimum number of cases to achieve the coverage without duplication by optimizing the test case combinations down to just the few required. Test engineers may devise models that are beyond the capabilities of the MBT tool – the tool simply chokes when given such a model. Unfortunately, scalability is something that is quite often ignored while running initial proof-of-concept projects which mean that organizations can easily invest on a tool that does not scale to real world industrial problems. Models must be simplified meaning reduced coverage and/or more manual effort is needed. This can be a very expensive and poor investment.

Conformiq has invested a significant amount of effort over the years to bring the performance of the test generation core engine to a level where we can manage real industrial problems. I wrote actually a short blog about this some time ago where I shed some light on what happens under the hood detailing how the tool can be for example deployed on a cluster or a cloud for fast test generation. It may seem easy to automatically split the model across multiple computation cores, but the real trick is to do it deterministically. This means that regardless of which cores and their order used, the generated test cases are always the same. I'm confident to say that Conformiq's test generation core is a magnitude of order stronger than any other tool on the market today.

Integrations with other tools and integration API's

The MBT tool is not best used as a standalone capability. Instead it should be tightly integrated with other tools used to document, manage and execute test cases. It is important to notice that most of the MBT tools do not actually execute the generated test cases but instead export them in to various different test execution environments. This is primarily because many of the MBT users have already invested in test execution infrastructure prior to moving to include automating the test design. Therefore, instead of replacing your existing test execution system when deploying MBT, you should look for a tool that integrates with your existing infrastructure. This same approach applies if manual test execution is employed; you look for a tool that integrates your ways of working.

Test execution infrastructure is again just one piece of the overall solution and the MBT tool should integrate with those other tools as well. These are tools like requirement management and test management systems, version control systems, and so on.

If there is no out-of-the-box integration with your particular tool, the MBT tool should offer integration API's that allow you to easily integrate with your tool. Investing in a tool that does not have proper integration API's can be risky as that can limit your freedom to upgrade your testing infrastructure in the future and support multiple test execution platforms.

Test review

Related to the quality of the tests, the tests need to be understandable as well so they should be very easy to review. You should not take it at face value that tools just magically create good quality tests and not spend any time on reviewing them - just the opposite. The tool should allow you to understand why every

test case is needed. Everything starts from “simple things” like having understandable and meaningful names and high level descriptions for test cases. For example, naming test cases such as “Test #1”, “Test #2”, and so on, does not really help the user to understand what the tests are all about, so manual test case renaming must occur.

In order to be sure that you have not missed anything in the test design, the tool must be able to convince the user that the generated test suite indeed meets all the requirements. That is, you must be able to assess the coverage of the test suite. You must be able to relate the tests back to the functional requirements and also to the model. You must be able to have tools for detailed analysis of each and every test case. You must be able to walk thru the test case step by step and simulate it against the model to gain full understanding of the tests, if that is deemed needed, and so on.

Model analysis and debugging

Since system models are human made, they may contain errors. A model that performs arithmetic can for example perform a division by zero while a concurrent model can have model-level thread scheduling that cause the threads to deadlock. The bigger and more complex the models get, the more important it is for the MBT tool to provide different means of analyzing issues in the model itself, simulate the model in order to gain understanding and identification of erroneous scenarios and even lack of coverage, and then to link the failed test execution results back to the model for better understanding the root cause of a failed test case. These again are items that often go unnoticed during the very first proof-of-concept pilots and again investing in a technology that does not support a full-fledged model analysis can be very short-sighted.

Conformiq Designer, for example, while performing the test generation, verifies that the model is internally consistent, i.e., the tool checks for the absence of internal computation errors (such as division by zero). If the model happens to contain an internal error, the tool will produce a comprehensive report that details the circumstances under which the problem occurred, graphically pin-points the problematic location in the model and shows a full execution trace to the problem. For further analysis of the problem, users can start a “model debugger” which is an infrastructure that allows the user to analyze various issues in the model and get a better understanding of the automatically designed and generated test cases. With these tools available, the model debugging and analysis is much faster and less error prone, and streamlines the whole modeling / test generation process. Debugging real world models can be the most time consuming part of deploying MBT. This becomes increasingly important if MBT is used in an agile development process.

Look for a solution that offers fluent model debugging and analysis capabilities.

Support and user community

System model driven MBT requires bit of a different skill set than traditional manual testing and some test engineers may feel slightly intimidated by these tools. Also, working with the models requires a different mindset than traditional testing, so some of the senior testers may feel alienated with these tools.

There are a lot of best practices collected over the years and lot of documentation available on how to address these problems. The MBT tool vendor should not only provide a state-of-the-art testing solution, but also provide training, documentation and best practices around these and other questions both before and during deployment.

Overall, PoCs are intended to do exactly what they say – Prove the Concept. They do not prove the capability for use on real programs. While this full testing cannot be done as a PoC, what can and should be

done is to select PoCs that expose the positives and negatives for all MBT tools being evaluated. Make the PoC really meaningful to you or run multiple PoCs to demonstrate tool differences and that it will work for you. It may be difficult to determine the real engine differences, but that understanding is THE most critical learning from running a PoC.

The author of this paper, Kimmo Nupponen, has been developing automated test design software for over ten years. He understands what is really needed for real world use and the differences between tool engines. He is the Chief Scientist at Conformiq.