CONFORMIQ

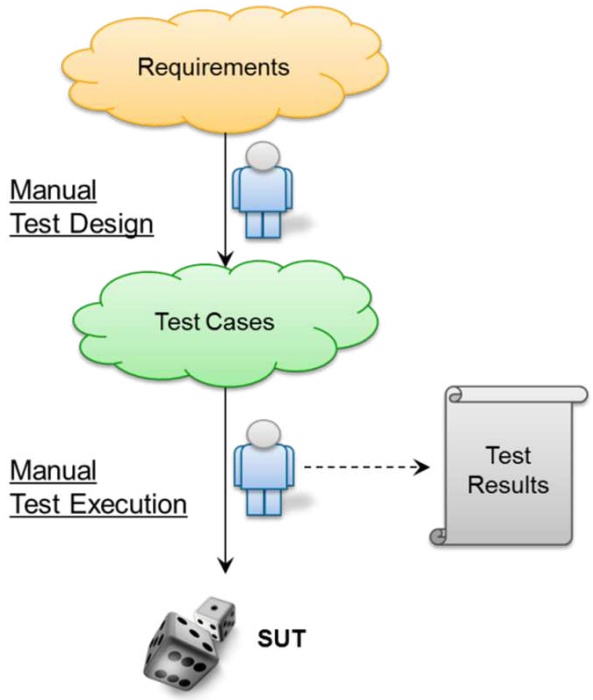
A Conformiq White Paper

Why Automate Test Design?

Traditionally the test automation has been focused mainly on automating the test management and test execution. Unfortunately, the test design often still remains a manual activity. The test design itself concerns making the decisions on (1) what to and what not to test, (2) how to stimulate the system and with what data values, and (3) how the system should react and respond to the stimuli. The test design is therefore a separate task from test execution and is done before executing the tests against the system. So even still today, automated tests are too often created and executed only for regression – not really to find defects in the new functionality. Traditional, manual test design and manual test execution are still prevailing approaches for testing new functionality. By automating also the test design, the testing efforts can be significantly reduced while at the same time the quality of the testing can be increased. One of the most promising approaches for automating test design is via a model based testing approach called system model driven test generation. This is the topic of this white paper.

MANUAL TESTING PROCESS

In order to see and understand why we need test automation in the first place, let's take a look at a completely manual testing process – this is the earliest form of testing but it's still widely used today



The *test design here is done manually* based on informal requirements documents. The test designer goes through the requirements document and manually invents test cases for testing an implementation that is based on the same set of requirements.

The output of the manual test design step is a document that describes the desired test cases. With the test cases, *test execution is done manually*. A manual tester follows the steps of the test cases and directly interacts with the SUT comparing the values of the SUT output with ones expected, finally recording the test verdict.

In order to carry out the test design, the test designer needs to possess expert knowledge about the SUT and he also needs to have test design strategy skills. The manual test execution requires less pure talent but what is needed is the ability to follow the steps of the test cases and knowledge about how to interact with the SUT.

The main benefit of the manual testing approach is that it's very easy to start with and the initial cost is low.

However, as everything is done manually, there are numerous shortcomings with this approach that can be divided into two groups, first the ones related to test execution and second related to test design.

When looking at shortcomings on the test execution side, the biggest and most severe issue is that there is no automated regression testing, meaning that we need to do the whole process over and over again when the system changes. This quickly becomes a boring and time consuming activity. This process is actually so costly and time consuming that it often forces teams to cut corners and sacrifice the quality of their work.

It simply does not scale, meaning that it forces an ever growing or at least the continuation of a costly manual head count intensive process that can be improved.

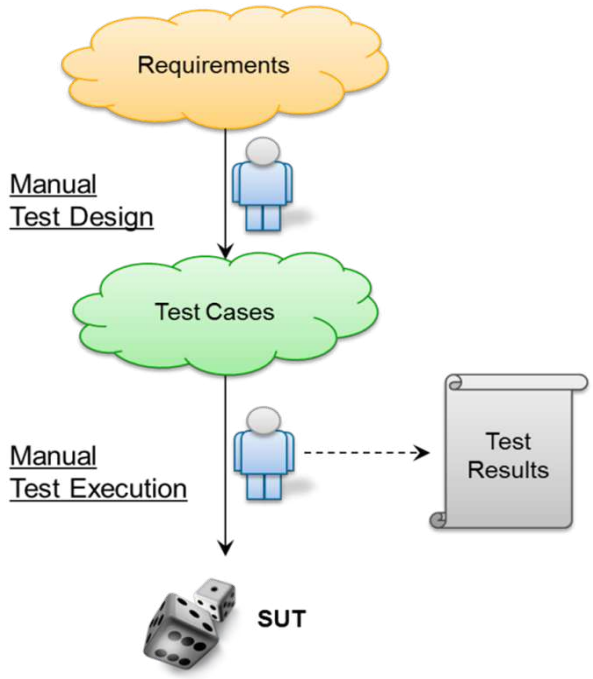
The second set of problems stems from the fact that test design is done manually which introduces great risks, - it's time consuming and hardly reproducible. We have a lot to say about this particular problem in this paper, and we will get back to this in greater detail a bit later.

Because everything is done manually, there is no systematic way to estimate functional coverage and therefore it is very difficult judge the progress of testing or quality of the produced test cases.

There is no automatic way of linking the requirements, therefore requirement traceability is either omitted or established manually.

RECORD AND PLAYBACK

Purely manual process can be improved by *automating test execution*. The record and playback method attempts to reduce the time and cost of test re-execution by *recording the interactions with the SUT during the first test execution session and then enabling a playback of the recorded test scripts* so that they can be re-executed at later time.



The initial test execution is a similar activity to the one in a completely manual testing process – the difference being that now we also record the interactions with the SUT. When the system changes, we actually have something that we can try to run against the system – the recorded test scripts.

As with the manual testing process, the record and playback approach is very easy to use and the initial cost can be low. As the interactions are recorded, one can trivially replay the recording allowing one to re-execute the test scripts for "free".

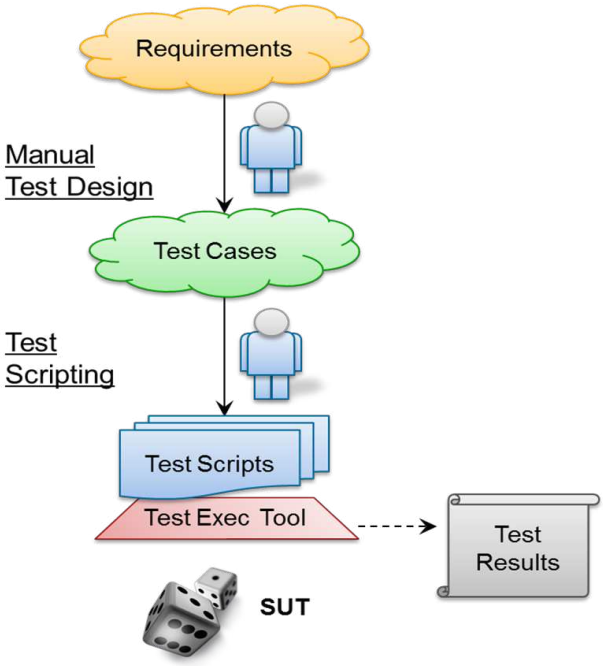
The main problem with applying record and playback to automate the re-execution of the tests is that it is *extremely fragile towards changes in the SUT*. This inability to adapt to small changes in the SUT often forces the test engineers to re-record all the test scripts when there is a small change in the SUT, causing a huge maintenance problem. The problem is so severe that these solutions are often abandoned after a couple of new revisions. Some of the record and playback tools try to alleviate this problem by enabling one to elevate the level of abstraction of the recorded test scripts by allowing one to make changes to them, for example, by using place holders in the recorded scripts which are then filled during the execution time from a data table.

As record and playback aims to only address the problem of re-executing the tests, it suffers from the same shortcomings as the manual testing process.

In practice, record and playback is not an attractive approach for addressing test automation delivering limited efficiency gains over manual.

SCRIPTED TESTING

In a scripted testing process, the *test execution problem is solved by automating it by writing test scripts*.



Instead of directly interacting with the SUT, the test engineer writes a collection of executable test scripts each containing one or more test cases. These test scripts can be automatically executed against the SUT. They stimulate the system with certain input values. Test scripts can be implemented in many scripting or programming languages and then executed on a framework that can read in scripts in that particular language.

The test execution tool records the output values, compares the observed values against expected values and finally gives out a test verdict. As test scripting is a programming task, test engineers need to possess different skills from test design and test execution.

As the test execution is automated using test scripting, one can already run the initial testing using the automated scripts. Regression testing can also be done for "free" by simply re-executing the test scripts.

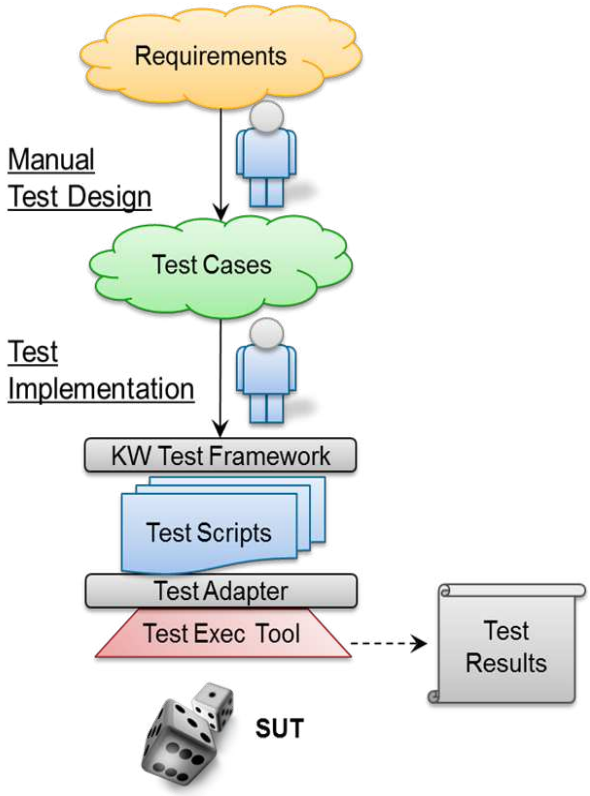
One of the biggest shortcomings of this approach is that the scripting in the first place is a complex activity and requires a lot of time and effort.

But what is even worse, is the maintenance problem that scripted approaches have. This stems from the fact that the test scripts need to be updated not only when the requirements change but also when some implementation detail changes. How much time and energy is then spent on maintenance depends on the abstraction level of the test scripts. Implementing nicely abstracted test scripts demands some advanced skills from the test designer.

As scripted testing focuses on addressing the test execution automation problem, it suffers from many of the same shortcomings as the manual testing process, namely from the risks and costs associated with the manual test design, ad hoc coverage and manual traceability.

KEYWORD DRIVEN TESTING

In order to overcome the maintenance problem introduced by scripted testing, the *abstraction level of the test cases can be elevated* using keyword driven testing.



The main idea here is to express the test cases in as an abstract form as possible while still providing enough details so that they can be readily executed against the real system.

In **data driven testing** or **data table testing** we have a set of abstract test cases that do not fix the data values but the data values are read from a data table during the test execution. This allows one to reuse same test scripts for testing the system with multiple different data values. This obviously will reduce the maintenance efforts.

In keyword driven or action word testing, one takes this concept a bit further and abstracts also the test steps in the test cases by introducing keywords or action words that then correspond to some well-defined fragment in the test scripts. This allows non-programmers to implement the test cases simply by constructing them using these action words, thus this enables them to work at a more concise and abstract level. The action words are mapped to actual test code by a keyword driven testing framework and the test

code needs to be implemented by engineers who can do programming.

The main benefits of keyword driven testing are that it allows engineers to work at more abstract and concise level and also that the tests can be implemented by non-programmers. As the keywords map to executable code fragments, keyword driven testing offers the same benefits as scripted testing, namely automatic execution of the test cases and automatic regression testing. An additional benefit is that the maintenance efforts are reduced compared to scripted testing because of the possibility of more reuse and abstraction.

However, the test data and test oracles are still designed manually. In addition the test coverage with respect to requirements and traceability, like with all the other approaches introduced so far, needs to be done manually.

THE NEXT STEP...

So what is then the next step? How should we solve the problems with the current test automation solutions? All these approaches rely on manual test design and therefore none of them guarantees a systematic and repeatable coverage of the system behavior. This non-repeatability is a huge risk already in itself. With manual test design it is really hard to assess the quality of your testing efforts which quite often leads one to evaluate the quality and progress of the manual test design process using spurious metrics such as number of test cases or number of hours spent on doing testing.

At the same time, manual test design is also a very expensive process - especially when there are changes in the requirements. In practice, test engineers are forced to manually analyze each of the test cases individually in order to see which test cases need to be updated, which removed and which added in order to fill the coverage gap when there are changes in the requirements. This loses a lot of productivity.

Finally, the requirement tracking in all of these approaches is done manually.

AUTOMATING THE TEST DESIGN

Traditionally in the test design phase, the test engineers and designers form an understanding of the system using the specification and requirements – they form a **mental model**. This mental model is *not one of tests but the system itself*. In a purely manual test design process, this mental model of a system is then turned into test cases in the mind of the test engineer. This is an implicit, creative process that is not really reproducible and is bound to the ingenuity of individual engineers. If you lack talent for doing good test design, you're out of luck.

As the test engineers form a mental model of a system, then it seems that test design can be automated by making this model explicit i.e. by expressing this mental model in a form that is understood by a computer and then generating test cases out of this explicit model.

MODEL BASED TESTING

Now when we have a computer readable model, we can apply **model based testing** to the problem of test design automation. Model based testing is currently a trendy thing and can mean 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 1) **graphical test modeling** approach, 2) **environment model driven** test generation., and 3) **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

The graphical test model is simplest of the approaches listed above and is actually nothing more than *modeling the test cases themselves in a graphical notation*. That said, graphical test case modeling aims to provide similar benefits to keyword driven testing i.e. by elevating the level of abstraction one can reduce the maintenance costs by enabling more reuse and increasing the productivity. The tools then turn these abstract graphical test cases in to executable test scripts.

The models that capture graphical test cases are easy to understand and the complexity to create one is low. Therefore the approach may appeal to non-programmers as graphical test case modeling does not require programming skills which is quite often expected for other model based testing approaches.

However, as we are modeling the test cases themselves, really the only thing that we are automating here is the creation of the executable test scripts. Therefore the value proposition is quite the same as with keyword driven testing. No test cases are created beyond what the modeler thinks of and when the design changes, the manual effort of remodeling is the same as the original effort.

Environment Modeling

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*.

For example, if we are testing an application running on a handheld mobile device, the environment constitutes the user who uses the device and the radio network. With an environment model you now describe how the environment – meaning the user and the network – operates with respect to the application including the details about testing strategies and output validators.

This style of modeling is quite near to tester's traditional thinking, after all these models essentially capture the operations of the tester. The models, however, are more complicated than simple graphical test case models because of the extra expressivity.

Because the test generation algorithms for environment models are well known and easy to implement, the tools are relatively robust and efficient. Because these algorithms are easy to implement, there are a lot of different tools available, both free and commercial, and companies often create even their own tools for generating tests from environment models.

These tools eliminate the need of manually writing the test scripts and some of the tools even allow you to annotate the model with requirement links thus enabling automatic tracking of requirements, which is a highly important and valuable feature. The fundamental problem, that is the test design, is still left as an exercise to the test engineer. The test engineer needs to manually describe the testing strategies and the test oracle, that is, the stimuli that we need to send to the system and the expected output from the system under test.

System Modeling

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*. This means that the system model is this mental model that the test engineers form while going through the requirements documentation now made explicit. The model describes how the system should work – not how it should be tested. For a moment, let's go back to our previous example about an application on a handheld device, where the application operates with a user of the application and the radio network. As opposed to other approaches, in system modeling we focus on the behavior of the application itself. We do not focus on how the user uses the device or how the radio network operates. We focus purely on the correct behavior of the application on the device itself. We model the behavior of the application, on a high level of abstraction, and then we leave the problem of test design to the computer. The

computer is then responsible of figuring out how the environment outside the application operates – the computer figures out how the user should stimulate the application, what kind of interactions we should see on the interface of the radio network and what kind out precise output the application should give to the user – the test stimuli plus what the test oracle. We do not model the environment; the computer figures that out from the model. Therefore, in case of system modeling, the *computer generates an environment that drives the real system*.

Creating a system model is more straightforward and a less error-prone process than modeling the test cases themselves or modeling an environment model. This is simply because the mental step involved in designing the testing strategies and oracles is simply omitted. Actually the modeling can be compared to a translation problem – the goal is to *translate the specification and/or requirement documents into computer readable format instead of creatively designing a highly complicated behavior for input selection and output validation* which are tasks that human mind is not really good at doing. Due to the fact that modeling is like encoding the requirements directly into the model without having to be creative, the model is very easy to update when the requirements change. The model is much easier to understand by stakeholders and use as reference for developers. This is a huge time saver in test maintenance.

There are two, somewhat contradictory goals when making a system model. The first is that the model should be smaller and more abstract than the real system – otherwise it takes too much time and money to describe one. It should focus on the key aspects that we want to test and should omit a lot of the details of the SUT. The second is that it needs to be accurate enough to capture the details that we want to test.

Comparison of the Methods

To quickly capture the similarities and differences between the three main approaches of model based testing, let's take a look at the table on the next page.

In system model driven testing, we model the correct and expected behavior of the system under test on a high level of abstraction which undeniably requires some technical skill. However, there is no need to design test inputs and outputs manually as they are automatically derived and generated. In graphical test case design, one models the test cases which makes modeling quite easy but offers no automation in input or output data selection. The user needs to do this design manually. Environment model driven approaches model the expected environment or the usage of the real system which again is technically more complicated task than for example the graphical test case design, while allows one to embedded testing strategies directly in to the model but still leaves the output validation as an explicit task for test engineer.

The requirement traceability is automatically created when using system model or environment model driven approaches provided that the model has been annotated properly with requirement links.

One of the fundamental differences of the three approaches is that only system models are **compositional**, meaning that only the system model driven approach allows one to construct a set of models that are combined together to form a model of a larger system. We will shed some extra light to this topic later in this presentation.

By applying the graphical test modeling approach, one can eliminate the task of writing test scripts manually like presented earlier. When adopting the environment model driven approach, one can also eliminate the task of establishing and maintaining requirement traceability links manually. However, only the system model driven approach eliminates the need of conducting test design explicitly plus test case maintenance. With other approaches these two tasks need to be done manually.

Finally, if we look at how the three different approaches work with projects that target not only one revision of the system but many, we see that the graphical test case modeling approach suffers from similar shortcomings that the traditional test automation solutions

	System model driven	**Graphical test case design**	**Environment model driven**
What is modeled	The correct behavior of the SUT on a high level of abstraction	The individual test cases	The testing environment and its logic
How input data is selected	Automatically	User defines it	A testing strategy—including input selection—is embedded as part of the model
How the test oracle (output validation) works	Automatically	Output data at execution time is compared to the output data predefined in tests	Explicitly implemented in the model
Technical complexity of models	High	Low	High
How tests are traced to requirements	Automatically	Manually	Automatically
Does it support composition	Yes	Usually no, because the actual concrete test data would need to match exactly	Usually no, because the testing strategies are not compositional
What tasks it eliminates	Design test cases Maintain test cases Write executable tests Maintain requirement traceability	Write executable tests	Write executable tests Maintain requirement traceability
What are the benefits over multiple release cycles	High: Model components can be shared and linked together Model maintenance is fast when requirements change	Low: Individual test cases can be shared (only) if they can be used as such Test maintenance focuses on individual test cases	Between the two other approaches: Testing strategies and oracles need to be maintained by hand

do. What you need to maintain are the individual tests. While the high abstraction level allows the same tests to be reused when there are small changes in the interface of the SUT, test engineers are forced to manually analyze each of the test case individually in order to see which test cases need to be updated, which removed and which added in order to fill the coverage gap when there are changes in the requirements. Test maintenance is a major concern with graphical test case modeling. At the other end of the spectrum we have system modeling, where the benefits of using system models are really high. This is because the individual model components can be shared and linked, therefore enabling **model reuse**, but also because changes to the requirements are really easy to reflect in to the model. Environment model driven approaches are there in the middle between these two extremes forcing the test engineer to maintain test strategies and oracles by hand.

SYSTEM MODEL DRIVEN MBT PROCESS

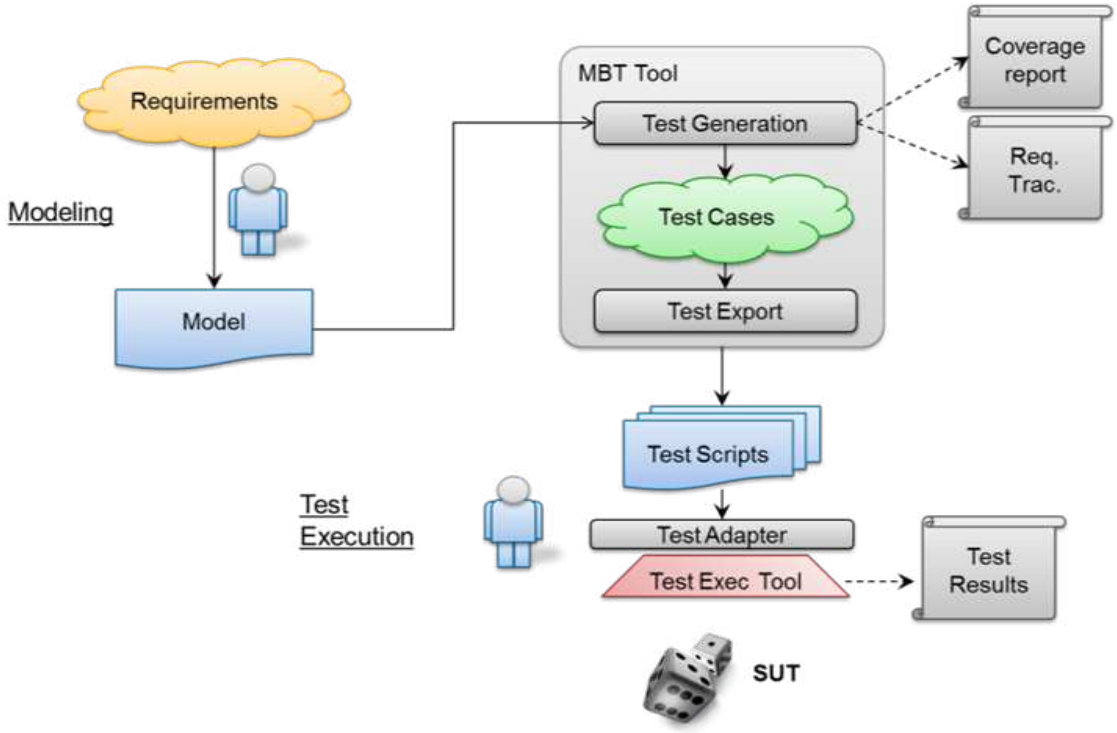
There are certain changes in the testing process that happens when system model driven MBT is taken into use.

First, instead of manually designing test cases, the test engineer writes an *abstract model of the SUT*. One essentially takes the specification or requirement document and encodes that in to a model which the test generation tool can understand. Typically this format is partially graphical and partially textual.

For example in the case of Conformiq Designer™, the model is defined using Java like textual syntax and optionally using UML state charts and class diagrams or alternatively activity diagrams. An important part of the modeling is to annotate the model with requirement identifiers to clearly show and document the relationship between the model and the functional requirements.

The next step, before we generate the tests, is the selection of test selection heuristics. This is an important part as there may well be an infinite number of possible tests for the tool to choose from. Therefore, we must state our goals and wishes for the test suite that the tool should produce.

Once the test selection heuristics have been defined,



one can generate test cases.

The output of the test generation is a collection of **abstract tests** which are sequences of operations from the model. The other two hugely important assets that are automatically generated are the **coverage report** and **traceability matrix**. The coverage report gives us valuable information about how well the generated test cases cover the model with respect to the coverage criteria that we selected. It's important to note that this coverage report is based on the model coverage, not the SUT. After all, we have not even executed the tests against the SUT at this point. The coverage report gives us information about the quality of the test suite and it also helps us identify model parts that are not well tested and covered. The traceability matrix, on the other hand, gives us the linkage between the model and the requirements.

The third step of MBT is to export and concretize the abstract test cases into executable and/or human readable formats. Often this happens via some translation or transformation tool. For example with Conformiq Designer™, you attach a *scripting backend* to your Conformiq project that then is used to export the abstract test cases in the desired format, whether directly executable or human readable documentation format.

The test execution happens using a test execution environment of your choice. In the case of manual execution, the abstract test cases are turned in to manual test plans and detailed test steps for manual test execution.

Finally the test execution results are evaluated using the test execution tool logs. An alternative approach is to import the test results directly back to the MBT tool so that the test execution result analysis can be done on the model level which makes it significantly easier and efficient to figure out the problem. This step is very similar to traditional testing processes and the goal is to determine the cause of the fault in a case of a test failure. The reason why the test fails may be because the SUT was implemented incorrectly, the model was crafted incorrectly or the requirements were incorrect in the first place.

COMPLEMENTARY SOLUTION

As the previous section suggests, MBT should not be seen as a competing solution with existing test automation solutions but more of a complementary one. As MBT aims to address the shortcomings of the more traditional approaches, it can leverage existing investments on test automation and can be really seen as an additional and highly valuable piece of the whole automation pipeline. MBT can be seamlessly integrated with existing processes and tools, both on the modeling side and test export backend side. On the modeling side, one can integrate with requirement management tools enabling one to check the completeness of requirement annotations in the model with respect to the requirements identified in the requirement management tool during the specification and requirement analysis phase. On the backend side, one has numerous different integration options with test execution tools, test management tools, and test documentation tools.

SYSTEM MODELING BENEFITS

We have already seen that the system model driven approach *relieves the user from designing, validating and maintaining individual test cases*. This stems from the fact that the test design problem is automated therefore allowing the user to focus on the correct behavior of the system, instead of individual tests.

Improved Quality

The first huge benefit is the improved quality of the test cases. This is because the automated approach to test design *lowers the risk of having incorrect, missed and redundant tests*. An engineer can, for example, accidentally miss a test case that is dictated by the requirements for example for an error handling case, a limit value of a data parameter, or an expiration of a rarely activated timer. The Aalgorithmic approach to test design eliminates randomly incorrect tests. There are fewer missing tests, because the algorithm does not accidentally miss corner cases. There are fewer redundant test cases because the resulting test sets are optimized rigorously by computer and checked for importance.

An important observation is that as the tests are always related to the requirements the quality of the generated test suite is always measurable.

Finally, the whole process itself is systematic and repeatable.

Improved Fault Detection

The core purpose of doing testing is to find flaws. The fault detection capabilities of MBT are *increased by lowering the risk of incorrect and missed tests*. The tools that implement the system model driven approach have been constructed so that they optimize the tests rigorously for coverage, non-redundancy and test efficiency.

The second aspect is the possibility to *generate different kinds of test suites for different purposes* that all target different aspects of the system operation. It suffices to select slightly different test selection criteria and let tools generate new test suites. All these features make MBT capable of producing a very good quality tests that are used to find defects that are difficult to find using other approaches.

This is also what we see in practice. Numerous practical experiences, case studies and proofs of concept, show that MBT is as good as or better in finding defects than manual testing. This is not surprising as when the system gets more complicated, the rigorous and comprehensive test design task becomes simply too overwhelming a task for the human brain. Computer is much better in this kind of endeavor.

Reduced Cost and Time

The time and costs can also be reduced by applying system model driven MBT. This stems from the fact that *creating a system model is straightforward and less error prone than describing the tests themselves*. The user makes the mental model explicit instead of inventing test cases based on that. This, increases the quality of the end result, but also reduces the time.

One model can also be used to generate multiple different test suites for different purposes. One essentially gets all the different test suites for free using a single model.

Time savings during the model maintenance phase really gets highlighted, because *model maintenance is significantly easier and more efficient than maintaining individual test cases*. We will talk more about maintenance aspect later in this presentation.

Finally, the test failure analysis is often easier and faster. One can, for example, inspect the path that the test took through the model in order to provide more understanding of the circumstances under which the problem was triggered. In some cases, it is even possible to import the test execution results back to the MBT tool for further analysis. Another thing is that MBT tools are often capable of generating the shortest possible path to the test failure, thus making the test analysis simpler. In addition, the tests are generated in a consistent fashion so the failure reports also tend to be more consistent. All this additional information makes it easier to understand the tests, the reasons for their failure, and most importantly to find and fix the problem.

Improved Traceability

Traceability is the ability to relate tests to the model, tests to the test selection criteria and tests to the requirements.

Requirements traceability means tracing your functional requirements through your system design and test. From the test design perspective this means that you should be able to *explain how your test cases and individual test steps are related to those functional requirements that have been articulated*.

Implementing requirements traceability has many benefits:

1) It helps to ensure that none of the functional requirements has been ignored in test case design.

2) It helps to explain tests and gives rationale why tests were generated. Requirement traceability helps in understanding tests, as the tests are linked to the

requirements they are supposed to test.

3) It helps in post-execution analysis of tests to pin-point which feature was actually malfunctioning.

Maintenance

The maintenance becomes an important factor for projects that targets not only a single revision of the system but many revisions. Traditionally when requirements change, a significant amount of effort is required to analyze and update the existing test suites. You need to go through every test case and see whether the test case and data are still valid, whether you should modify them in some way, or whether you should eliminate the test altogether. In addition, you need to decide whether you need to introduce new tests for bridging the coverage gap and with what kind of test cases.

With the system model driven approach, the maintenance efforts are significantly reduced. This is because the *model is typically smaller than the test suites and because the requirement updates can often be easily reflected into the model*.

After we have made the updates to the model, a new test suite can be automatically generated. When regenerating the test suite, the tools establishes an incremental traceability and can directly report which of the test cases were removed, which were added and which became redundant.

Prospect of Reuse

Related to the model maintenance, one of the benefits of system model driven testing is derived from the ability of reuse. Reuse, also in the context of test generation, *offers great rewards by allowing one to save time and money by reducing the amount of redundant work*.

The possibility for reuse exists because *system models are* **compositional** *and because system models are often expressed with languages that offer direct support for reuse*. For example, Java like notation of Conformiq Designer™ allows one to reuse models via concepts familiar from object oriented paradigms such as inheritance, delegation, communication and parameterization.

Model composition is an important feature really only available with system models because it allows one to reuse the same models for generating *function, component, system and end-to-end tests*. Model composition means that you can take multiple smaller models and combine them into one bigger model. This allows you to first model and test smaller features independently and then later combine the models and then test that the features also work as expected when combined together.

Model composition in addition enables early detection of **interoperability issues** where components, even if independently operating correctly, don't work correctly when connected together. Interoperability can be tested essentially for free when one has the models of the components to be connected together.

Improving Requirements

Finally, one quite unexpected benefit of model based testing is that the mere act of modeling the system behavior often improves the quality of the requirements. That is a lot of defects can already be spotted in the model of the specifications and requirements before even writing a single line of code. The requirements often contain ambiguities, omissions and contradictions. As one writes a model of the system behavior, one often raises a lot of questions regarding the requirements, so already the modeling process can exposes a lot of issues with the requirements.

Actually when you think of it, this should not come as such a surprise. After all, system modeling involves the development of small high-level prototype of the real system and it has been long known that prototyping is a good and efficient way of finding requirement bugs.

BENEFITS DO COME WITH A PRICE…

As with any disruptive new technology, there are some obstacles that hinder deployments. These obstacles, luckily enough, can be overcome by training and experience.

The first practical issue is that system modeling requires a different skill set than manual test design. System models are abstract small programs, therefore the test engineer must be able to abstract and design programs and this requires programming skills. There are ways of minimizing the amount of "coding" that one needs to do in order to craft a model but the fact is that all the real applications are computational processes and, with what we currently know, the only known efficient way of describing a computational process is in terms of a programming language. This directly implies that the real system models are small abstract programs themselves defined using a programming language. But one should not think of this as a shortcoming or a disadvantage of the solutions. It gives you a very powerful way of describing your system in a concise and sound fashion. And also quite often specifications and requirements are written in an informal notation that can be quite naturally translated in to program or model code – take business rules as an example. They are quite often described in pseudo code, decision tables or trees. Or take a protocol specification – you see a lot of state charts and pseudo code fragments, all which can be quite easily translated into "code".

Another thing is that test engineers may feel alienated modeling the system behavior as that does not involve the same thinking process as you typically have when doing testing. You don't really think about testing when you are modeling so, in a sense, the role of the tester moves a bit closer to the developer or designer role. This, especially with senior test engineers and designers who have worked for long time on more traditional approaches to testing, manifests itself in such a fashion that they will use system model driven approaches to capture the test scenarios and test cases themselves, instead of modeling the expected system behavior. There is nothing wrong in using the tools in this way, per se, but one would gain more of the benefits by trying to adjust to the new way of thinking and to the paradigm shift that system modeling introduces, ultimately by modeling the correct and expected system operation instead. Otherwise, one may not experience the great benefits that system modeling has to offer and one needs to settle on only those benefits that, for example, environment modeling approach has to offer.

A pragmatic issue that test engineers may run into is the limitation of the tools themselves. This is because test generation from system models is computationally an extremely difficult task, therefore the test engineers may devise models that are beyond the capabilities of the tools – the tools simply choke when given such a model. Therefore, in certain cases the test engineers may need to gain extra knowledge about the tools and the algorithms that they apply in order to figure out how to avoid developing a model that kills the tool. Scalability issues are something that, for example, Conformiq takes very seriously and constantly invests a lot into research and development of more efficient algorithmic approaches to automated test design. But, here again, one should remember that as test engineers and designers already have some mental model of the system's correct operations in their minds, this means that *constructing the test cases is very difficult for a human also*. And even worse, the human mind is horrible in this kind of creative, combinatorial exercise. Thus it is beneficial to model the system behavior.

But there are other factors as well.

Especially when test engineers are exposed to such a technology for the first time, they may be skeptical about the tool capabilities: Can a computer really design tests as good as I can? Our systems are so complicated that is it really possible to automate the test design? These questions can only be answered by taking a deep dive and see what the tools can deliver. Practical experiences from numerous industry segments where the system modeling approach has been applied to vastly different problems show that they can. They really can. It's not going to take away your job. Instead it helps you to focus on more important things and makes you more productive.

But it is also a leap from the comfort zone for the project management. With new philosophy and technology in place, you need to adjust your ways of planning but also tune the way that you measure progress. There are a lot of best practices collected over the years and lot of documentation available on how to address these problems. Most of the tool vendors provide trainings around these areas and are more than happy to share some of their best practices.

CONCLUSIONS

As we have seen, traditional test automation focuses mainly on two aspects; test management and test execution. With these solutions, the process of test design, that is the process of deciding how to test, what to test and what not to, is left as a manual activity. We have seen that manual test design introduces a lot of risks and it takes a lot of time, especially when the requirements change.

System model driven test generation is an effective and complementary way of addressing the shortcomings of existing test automation.

First, it automates the design of functional test cases to reduce the design cost and to increase the quality.

Second, it reduces the maintenance costs of the tests.

And third, it automatically generates coverage reports and traceability information from requirements to the tests and back.

System model driven test generation offers significant benefits in terms of improved quality, improved SUT fault detections, improved traceability, improved maintenance, improved model reuse, reduced cost and time, and improved requirements.

MBT is a more sophisticated approach to testing than earlier generations of testing tools. Operating with these tools requires a different mind and skill set than more traditional testing tools. However, the practice shows that these hurdles can be overcome by proper training and experience plus a willingness to make the change succeed.

And once you pass these initial hurdles and you start to see the benefits, you really do not want to go back.

ABOUT CONFORMIQ

Originally established in 1998, Conformiq is a leading solutions provider for automated test design and advanced model-based testing, dedicated to improving test design processes within software-intensive product companies operating in business-, mission- and life-critical industry segments.

Conformiq Designer™ is the company's fourth-generation test design tool, built upon a decade of advanced basic and applied research as well as testing and test design experience.

Privately held, independent and known for extraordinarily responsive customer service, Conformiq is the partner of choice for companies who are ready to step ahead of the curve.

For more information about Conformiq and the company's software and services, please visit www.conformiq.com.

www.conformiq.com

12930 Saratoga Ave Ste B9
Saratoga, CA 95070
USA

Tel: +1 408 898 2140
Fax: +1 408 725 8405

Westendintie 1
02160 Espoo
FINLAND

Tel: +358 10 286 6300
Fax: +358 10 286 6309

Stureplan 4C
SE-11435 Stockholm
SWEDEN

Tel: +46 852 500 222
Fax: +358 10 286 6309

Maximilianstrasse 35
80539 Munich
GERMANY

Tel: +49 89 89 659 275
Fax: +358 10 286 6309