CONFORMIQ
Automated Test Design™

A Conformiq White Paper

Comparing Automated Test Design Methods

How automatic is your automated test design process? There are three primary methods used in automated test design tools. They all deliver improvements in the test design process, but there are significant engine differences that you should fully understand prior to selecting your tools. In this paper we will compare and contrast these methods and discuss the limitations and benefits from each.

The three approaches to automated test design all begin with different approaches to modeling. The perspective used may not seem much different or very important, but it is critical to the internal test design engines used in the different automated test design tools. The type of modeling dictates the tool's ability to convert that model into test cases and many associated capabilities that are important to deliver the maximum user benefits.

Comparison of the Methods

The three types of modeling methods for automating test design are ***system model driven***, ***graphical test case design***, and ***environmental model driven***. To briefly capture the similarities and differences between the three main approaches of model based testing, take a look at the following table. Key features that highlight important capabilities needed to deliver maximum benefits from the transformation to test design automation are listed and the differences each approach delivers are compared.

The more automated or, to spin the word slightly to allow you to better differentiate between the uses of the term automation, the more *automatic* the tool's capabilities, the greater are its benefits.

	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

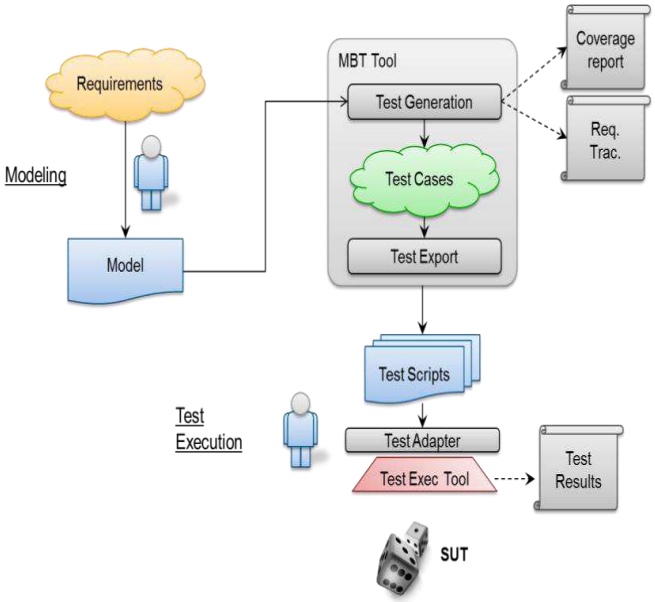
How the test oracle (output validation) works	Automatically	Output data at execution time is compared to the output data predefined in tests	Explicitly implemented in the model
Technical complexity of models	High	Low	High
How tests are traced to requirements	Automatically	Manually	Automatically
Does it support composition (combining and reusing model components)	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 exactly reused Test maintenance focuses on individual test cases	Between the two other approaches: Testing strategies and oracles need to be maintained by hand

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 for they are automatically derived and generated. In graphical test case design, one models the test cases, which makes modeling easy, but offers no automation of 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 is also a more technically complicated task than, for example, the graphical test case design, which allows for the direct embedding of testing strategies to the model, but still leaves output validation as an explicit task for test designer.

Requirement traceability is automatically created when using system model or environment model driven approaches, provided that the model is 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, the task of writing test scripts manually can be eliminated. When adopting the environment model driven approach, the task of establishing and maintaining requirement traceability links is manual. *Only* the system model driven approach eliminates the need for conducting test design explicitly and test case maintenance. With the 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 as the traditional test automation solutions. The individual tests need to be maintained. While the high abstraction level allows for the same tests to be reused when there are small changes in the interface of the SUT, test designers are forced to manually analyze each of the test cases individually in order to see which test cases need to be updated, which need to be removed, and which need to be added in order to fill the coverage gap when there are changes in requirements. Test maintenance is a major concern with graphical test case modeling. At the other end of the spectrum is system modeling where the benefits of using system models are high. This is because the individual model components can be shared and linked, enabling **model reuse**, but also because changes to the requirements are easy to reflect in the model. Environment model driven approaches are in-between these two extremes, forcing the test designer to maintain test strategies and oracles manually.

System Model Driven MBT Process

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

First, instead of manually designing test cases, test designers write *an abstract model of the SUT*. They essentially take the specification or requirement document and encode it into a model that the test generation tool can understand. Typically this format is partially graphical and partially textual.

For example, in the case of Conformiq Designer™ used often for testing embedded software, the model is defined using Java-like textual syntax and optionally using UML state charts and class diagrams or using the new Conformiq Creator™ modeling option used for testing IT and enterprise software; activity diagrams and interface actions are used to define the model. 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 most advanced MBT tools allow import of test cases saved from record and playback execution or from existing manual test cases themselves. The import from recorded test cases accelerates model creation while model generation from manual test cases requires some additional manual effort to normalize the tests and eliminate duplicates. This capability enables brown-field projects to fully leverage MBT benefits.

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

Once the test selection heuristics have been defined, test cases will be automatically generated.

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

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

Test execution happens by 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, 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 more efficient to figure out the problem. This step is similar to traditional testing processes in which 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. The tester needs to decide the cause.

Complementary Solution

As the previous section suggests, MBT should not be seen as a competing solution with existing test automation solutions, but as more of a complementary one. Since MBT aims to address the shortcomings of the more traditional approaches, it can leverage existing investments of test automation and can be seen as an additional and highly valuable piece of the entire SDLC automation pipeline. MBT can be seamlessly integrated with existing processes and tools, both on the modeling and test export backend sides. On the modeling side, requirement management tools can be integrated to enable checking for 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 phases. On the backend side, there are many different integration options with test execution tools, test management tools, and test documentation tools.

System Modeling Benefits

The system model driven approach *relieves the user from designing, validating and maintaining individual test cases*. This is due to the fact that the test design problem is fully automated, allowing users to focus on the correct behavior of the system, instead of on many individual tests.

Improved Quality

The first benefit of automating test design is the

improved quality of the test cases. The automated approach to test design *lowers the risk of having incorrect, missed and redundant tests*. A test designer or engineer can accidentally miss a test case that is dictated by the requirements, for example with an error handling case, a limit value of a data parameter or an expiration of a rarely activated timer. The algorithmic (system modeling) approach to test design eliminates randomly incorrect tests. There are fewer missing tests because the algorithm does not accidentally miss corner cases. With this modeling approach there are also fewer redundant test cases because the resulting test sets are optimized rigorously by the computer and checked for their importance.

As tests are always related to the requirements, the quality of the generated test suite is always measurable and what is not covered is known.

Finally, the whole process itself is systematic, consistent, and repeatable adding more benefits.

Improved Fault Detection

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

The second aspect is the ability to *generate different kinds of test suites for different purposes* that all target different aspects of the system operation. It selects slightly different test selection criteria and lets tools generate new test suites. All these features make MBT capable of producing very good quality tests that find defects that are difficult to otherwise 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 than manual testing in finding defects. When the system gets more complicated, the rigorous and comprehensive test design task becomes too overwhelming a task for the human brain and computers are much better for this effort.

Reduced Cost and Time

Applying system model driven MBT can also reduce the time and costs. 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 it. This increases the quality of the end result while also reducing the time.

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

The time saved during the model maintenance phase is particularly important because *model maintenance is significantly easier and more efficient than maintaining individual test cases*. We will talk more about maintenance aspect later in this paper.

Scalability issues are something that Conformiq takes very seriously and invests a lot of time into the research and development of more efficient algorithmic approaches to automated test design. The key need here is to speed the test generation time since you cannot tell if you have good tests and proper coverage until the test cases are generated. If the test cases are selected manually or modeled as use cases, the engine needs little time because the user has already done the test optimization work.

However, if you are using the system model generation approach used by Conformiq, the design selection is fully automatic without user intervention. Thus the engine needs to do the heavy lifting of determining the optimal minimum number of test cases to cover all test points using the selected test design algorithms. To accomplish this and effectively handle large (read “real world”) models, Conformiq has designed its engine to automatically split a model

across all available processors, be they local or on multiple servers, to deterministically generate test cases quickly. The splitting is easy – making it deterministic is hard and Conformiq is unique in delivering this important capability.

The importance is quickly understood by an example. Since correct test generation is based on the model, the model must be changed and test cases regenerated until it is accurate and complete. This is an iterative process that may take many revisions. If each test generation takes one hour rather than 10 minutes, it is easy to recognize the efficiency gained from multicore processing.

Finally, the test failure analysis is often easier and faster with system model driven MBT. For example, the path that the test took through the model can be inspected to provide more understanding of the circumstances under which the problem was triggered. In some cases, it is possible to import test execution results back to the MBT tool for further analysis. MBT tools are also capable of generating the shortest possible path to the test failure, making the test analysis simpler. In addition, tests are generated in a consistent fashion so the failure reports are also more consistent. 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

This means tracing functional requirements throughout system design and test. This test design perspective allows for the *explanation of how 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 ensure that none of the functional requirements have been ignored in test case design.
2) It helps explain tests and gives rationale as to why tests were generated. Requirement traceability helps in understanding tests because tests are linked to the requirements they are supposed to test.
3) It helps in post-execution analysis of tests to pinpoint which feature was actually malfunctioning.

Maintenance

Maintenance becomes an important factor for projects that target 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 existing test suites. It is necessary to go through every test case and see whether or not the test case and the associated data are still valid, whether they should modify them in some way, or whether they should be eliminated altogether. In addition, it is necessary to decide if new tests for bridging the coverage gap need to be introduced or not and with what kind of test cases.

With the system model driven approach, 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 the updates to the model have been made, a new test suite can be automatically generated. When regenerating the test suite, the tools automatically establish an incremental traceability and directly report which of the test cases were removed, which were added, and which determined to be 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 saving 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 for the reuse of models via concepts familiar from object-oriented paradigms, such as inheritance, delegation, communication, and parameterization.

Model composition is an important feature that is only available with system models because it allows for the reuse of 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 test that the features work as expected when combined together. It also allows for reuse of models as reusable testing IP, thus enabling a model of the system or application to be easily and quickly changed to match new requirements or customers.

Model composition also 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 the models of the components are connected together.

Improving Requirements

Finally, one possibly unexpected benefit of model based testing is that the mere act of modeling the system behavior often improves the quality of the requirements. A lot of defects can 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, many questions regarding the requirements are raised, so the modeling process can expose a lot of issues with the requirements.

This should not come as such a surprise since system modeling involves the development of a small high-level prototype of the real system and it is known that prototyping is an efficient way to find requirement bugs.

Benefits Come with a Price…

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

The first practical issue is that system modeling requires a different skill set than manual test design. System models are abstract representations of the system operation in an executable form. With Conformiq Creator, the test designer or SME needs to understand the system but only needs to use existing components and connect them together to create the model. However with Conformiq Designer the models are really small programs, so the test designer must be able to abstract and design programs, which require programming skills. There are ways of minimizing the amount of "coding" that is needed to craft a model, but highly complex embedded software applications are computational processes and, the most efficient way of describing a computational process is in terms of a programming language. This should not be seen as a shortcoming or a disadvantage because it provides a powerful way of describing a system in a concise and sound fashion. Often specifications and requirements are written in an informal notation that can be naturally translated into program or model code. As one example, business rules are often described in pseudo code, decision tables, or trees. For another example, take a protocol specification, which also contains many state charts and pseudo code fragments, all which can be quite easily translated into "code" for a Designer model. AUTOSAR specifications are state charts with English notation making them very straightforward to model.

Test designers may feel alienated when modeling system behavior because it does not involve the same

thinking process that they are used to. 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 a way 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 with using the tools in this way, but there are more benefits in adjusting 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, the great benefits that system modeling has to offer may not be realized and one needs to settle on only the more limited benefits that the environment modeling approach delivers.

A pragmatic issue that test designers run into is the limitation of the tools themselves. This is due to the fact that test generation from system models is an extremely difficult task, therefore the test designers and engineers may devise models that are beyond the capabilities of the tools and the tools simply choke when given such a model. Therefore, in certain cases test designers may need to gain extra knowledge about the tools and the algorithms that are used in order to figure out how to avoid developing a model that kills the tool.

Conclusion

All test design automation tools deliver benefits. However, when making the significant transformation from manual test design to using automated test design tools, it is the best time to decide the future of your testing process and select the most appropriate tool that best meets your near and long term needs.

Of all the different test design automation methods, 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.

Thus environment modeling *helps automate* the test design generation while system modeling *automatically* generates the test design. The impact of this difference becomes quite large in practice.

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 "under the hood" differences between MBT tool engines. He is the Chief Scientist at Conformiq.

www.conformiq.com

4030 Moorpark Ave
San Jose, CA 95117
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 659275
Fax: +358 10 286 6309

29 M.G. Road Ste. 504
Bangalore 560 001
INDIA

Tel: +91 80 4155 0994

v0715