ICT GROUP

Model-Based Testing: Improved Quality-Assurance with less effort

ICT High Tech - Center of Excellence

Abstract

Model-Based Testing (MBT) is a novel testing approach that automates exploratory testing. By automating, this process can be done faster, more systematically, and fully automated, for example during downtime at night. This paper will describe how model-based testing and its tools work, and how we successfully applied it to automatically test our system. We believe the addition of MBT to your test process can make a valuable contribution to the quality of your product.

1 Problem Statement

Quality-Assurance is a key component in the delivery of any kind of machinery, especially for equipment that can potentially harm humans, is expensive to repair, or is vital to have low downtime on. We are currently seeing several trends in the hightech systems engineering field: systems are becoming more and more complex, thus needing more and more quality-assurance effort; At the same time, it is becoming increasingly difficult to have hardware available for testing, due to cost or availability of parts. Lastly, it is becoming increasingly difficult to find skilled employees to meet the increasing need for quality assurance. Since it is not possible to throw more resources at the problem, it is therefore key to improve the efficiency of the QA process, to ensure we can continue to deliver safe, reliable, and performant systems.

2 Methodology

As a potential solution to improve efficiency, we propose the use of Model-Based Testing (MBT). We will compare it to 'traditional' testing to show how it is different, and explore its strengths and weaknesses.

2.1 Traditional Testing

Traditional testing starts by making a test-plan of some sorts. This test plan can be a document of test steps and expected results to be manually executed and checked off, or it can be a series of fully automated (possibly BDD) acceptance tests. In reality it is usually a combination of various techniques: a suite of automated tests in combination with a manual QA sign-off. What these processes have in common is that the test plan needs to be created by hand: a developer or QA person has to imagine all the possible uses of the system, and make sure the important scenarios are covered by the tests in the test plan.

The advantage of this approach is that you can be sure the important scenarios and use-cases are covered by tests. This makes it so that when the test plan passes, there can be a high confidence the system is functioning correctly. The disadvantage of this approach is that it requires a lot of up-front work to come up with every test scenario, and to describe all of them. Furthermore, the number of possible paths through the system scales exponentially with the complexity of the system. If the system becomes sufficiently large, the test suite will become even larger, greatly increasing required maintenance effort. This is the problem model-based testing proposes to solve.

2.2 Model-Based Testing

In contrast to traditional testing, model-based testing does not require writing out every test scenario. Instead, a model of the system's interface behaviour is created. This model describes the different states of the interface/system, and the ways in which the system transitions from one state to another. This treats the system under test as a black box: internal functioning is hidden, only the interfaces to the outside world are known.

2.2.1 Interface models

Interfaces are modelled as labeled transition systems, also known as automata or state machines. This means that it distinguishes various states of the interface/system, and specifies the way in which this system can transition from one state to the other: either as a result of a stimulus from the outside world, or spontaneously. In the example Figure 1 below, the behaviour model of an automatic door is given. This state machine has two 'stable' states: *open* and *closed*, and two transient states: *opening* and *closing*. Stimuli to the system from outside are prefixed with a ?, and responses from the system to the outside world are prefixed with a !.

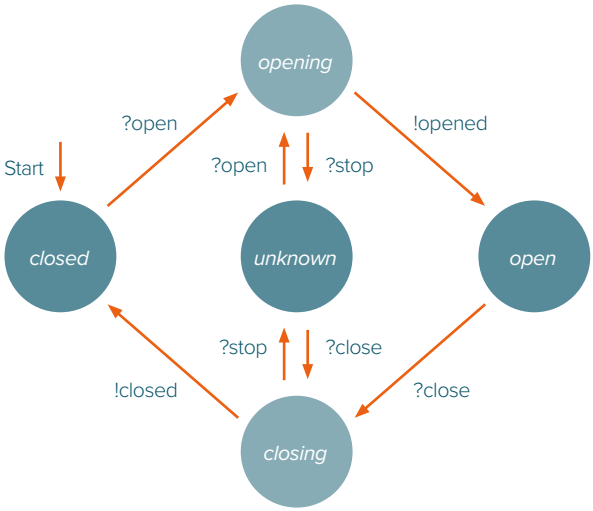


Figure 1: Example model of an automatic door

This model expresses the following behaviour: The system starts in the closed state. Whenever a *?open* stimulus is given, the door goes to the *opening* state, from which it will at some point send a *!opened* response, letting us know the door is now in the *open* state. From there, if a *?close* stimulus is given to the system, the door goes into the *closing* state. At some point then the door will send a *!closed* response, letting us know it is now back to the *closed* state. At any point during *opening* or *closing*, the stimulus *?stop* can be triggered. This will cause the door to stop in an *unknown* state, and wait for either a ?*open* or *?close* stimulus.

Note that this model only describes the externally visible behaviour of the system interface. No details are given about how the door is opened or closed. The internal behaviour such as which sensors/actuators are used or in what sequence the different parts of the door move, is irrelevant to the behaviour of the interface, and treated as a black box.

2.2.2 Model-Based Test execution

To test the example system in Figure 1 using traditional testing, you would have to create a test for every possible walk through the system. This means you would need to create a list of test scenarios similar to:

- ?open → !opened → ?close → !closed
- ?open → ?stop → ?close → !closed
- etc...

In contrast, with model-based testing, the idea is to let the test application come up with these scenarios, based on the model. This means there is ***no*** need for individual test scenarios. Instead, the interface model is the only input needed. Based on the provided mode, the test tooling will walk through the model, and try to provoke stimuli it sees available, while waiting for system responses the model claims to come.

2.2.3 Non-determinism

You may have noticed in the example model Figure 1 there are some states where multiple stimuli (starting with ?) are possible. This means that according to the model, one of these stimuli can be triggered nondeterministically, i.e. we cannot predict beforehand what will happen. While the test is running, one of the possible stimuli is chosen to execute semi-randomly (the reasoning behind this will be explained later). Therefore, it is not possible to predict beforehand which branch of the behaviour will be chosen during each test run.

Similarly, there can also be states in the model where multiple responses (starting with !) are possible. This indicates that one of the possible responses will be fired non-deterministically. That is, we can be sure one of these responses will be fired, but we cannot predict which one. During the test that means the tester will wait in the state for the first seen response of the modelled possibilities, and then continue to the next state as indicated. This means only one of the possible responses is expected to arrive; if multiple are expected to arrive, that would need to be explicitly described in the model. If a response is received from the system that is not expected according to the model, or if no response is received at all (within the timeout), the test will fail.

2.2.4 Coverage and model-exploration

In the previous section we discussed that due to nondeterminism, the test will semi-randomly walk through the behaviour model during the test. In reality, doing this purely randomly would be a waste of time. A smarter way to do this would be to consider model coverage: while testing, the test application can track which nodes and transitions it has already seen. This will result in some kind of percentage-of-the-model-seen while testing, as a measure of the testing progress.

This can usually be visualized like shown in Figure 2:

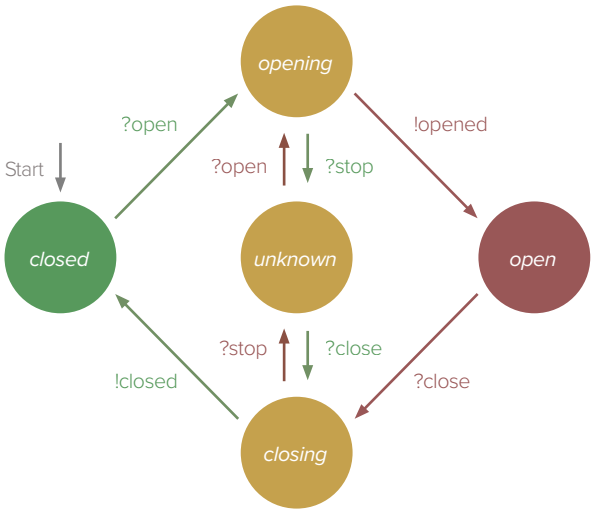


Figure 2: Example coverage of a test run

Next to being a convenient visualisation of which part of the model was seen during the test, and which part was not, this measurement of coverage also provides confidence in the test outcome. Similar to more common forms of coverage such as code coverage, a high coverage percentage gives a good confidence about the completeness of a test suite. Getting 100% coverage indicates the test has tried all possible stimuli, and has observed all expected responses (according to the model). This does not mean the system cannot still behave in a way that is outside of the expected model, but it at least shows that it has not done so yet.

2.2.5 Intelligent test execution

Using the coverage information of a test run, the test exploration can be made more intelligent. Instead of randomly choosing one of the possible stimuli, the test application can make choices that attempt to maximize the local or global coverage of the model. This means that in a node where the tester has previously already triggered two out of three stimuli, it can choose the third – uncovered – stimulus to increase the test coverage. Even in cases where all stimuli local to a node have already been seen, the tester can intelligently pick the one that has the highest probability of getting the system into a part of the behaviour model that was not yet seen during the current test run.

3 Practical Application

For the remainder of this document we will demonstrate model-based testing as implemented with Axini[1], a commercial tool for model-based testing. Axini is ***in the loop*** while running tests. This means that it does not generate all test scenarios up front, but instead generates the next step in the test, while the test is running. This saves a lot of disk space and compilation time required for pre-generated tests, and it also allows it to make intelligent decisions based on how the system is reacting to the test.

3.1 System under test

For demonstration purposes, this paper will show modelbased Testing as applied to our demonstrator system. This is a warehouse that stores disks of different colours in boxes. Items are taken from and placed back into boxes at the output location, but the boxes themselves never leave the warehouse. The interface is kept simple and limited to initializing, storing, and retrieving items from the warehouse. The system is shown in Figure 3.

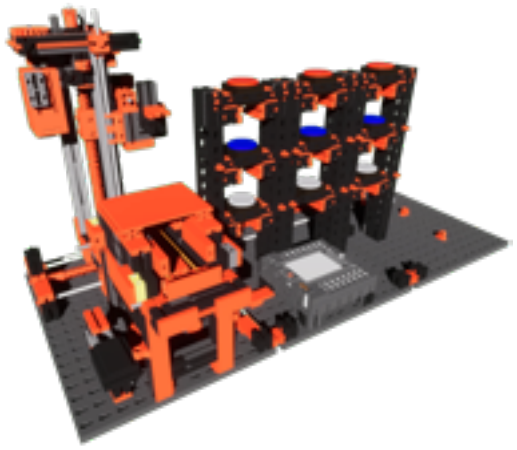


Figure 3: The digital twin of the warehouse to be tested. Note the storage rack on the right, stacker robot in the back and import/export conveyor on the left.

3.2 Test model

The state transition diagram of our system behaviour is shown in Figure 4:

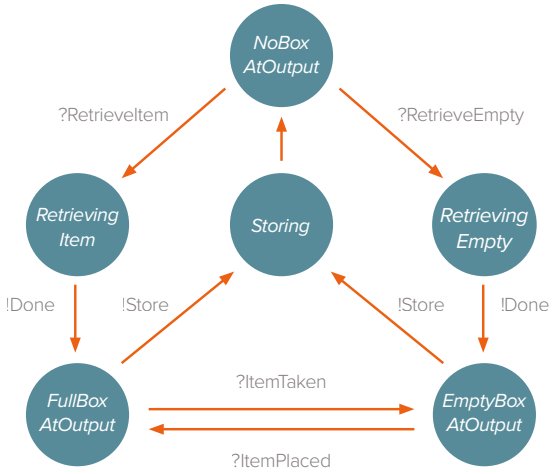


Figure 4: behaviour model of the warehouse

The model of our system identifies three stable states:

- *NoBoxAtOutput* – There is no box at the output. Allowed actions are: *?RetrieveItem* to retrieve a filled box, and *?RetrieveEmpty* to retrieve an empty box.
- *FullBoxAtOutput* – There is a box at the output with an item in it, that can be removed externally. Allowed actions are: *?Store* to store the box again, and notification *?ItemTaken* notifies the warehouse the item was removed from the box, and the box is now empty.
- *EmptyBoxAtOutput* – There is an empty box at the output, it can accept an item being placed into the box externally. Allowed actions in this state are: *?Store* to store the empty box, or notification *?ItemPlaced* to let the warehouse know the box is now filled.

In between these stable states, there are some temporary states: *RetrievingItem*, *RetrievingEmpty*, and *Storing*. In these states, the system is busy fulfilling its request and will reply with a *!Done* event once done. Note that during these temporary states, no actions are allowed.

3.3 Adapter

Since Axini[1] (and also other model-based test tooling) provides a generic interface to run the test, an adapter is required to map the generated steps from the test protocol to our system-specific interface.

In case of Axini, it is running on a server (local or in the cloud), and sends basic strings to indicate what to do next. We created a test adapter application in C# that is running together with our system-under-test, and connects to Axini via a websocket. It then translates the stimuli received from the test runner to calls on the gRPC interface of our example system. Conversely, it also transforms responses from the system into the test protocol, and notifies Axini about them. Figure 5 shows a schematic represenation of this setup:

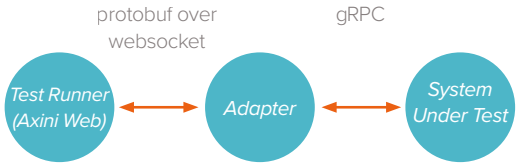


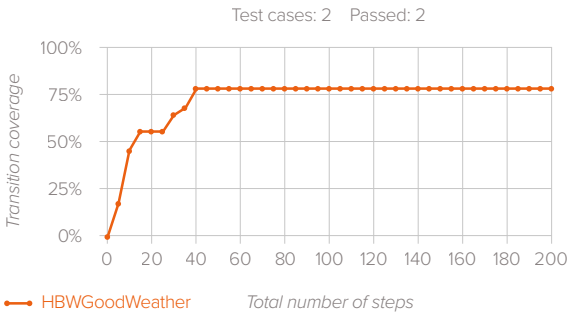
Figure 5: Test adapter setup

The adapter allows the test runner to focus on its task, generating test scenarios, without needing information about the system it is testing. The knowledge holder of the system-under-test is then responsible for translating the general test steps to calls on the system they are the experts at. This allows a great deal of flexibility: The test runner can run anywhere, as long as it can be connected to; similarly, the adapter can also run anywhere, as long as it can connect to the test runner and the system-under-test. It also means freedom of technology for the adapter. We are running an adapter written in C# that connects to our gRPC interfaces, provided by a C++ application, but we could also have chosen to write the adapter in for example Python or C++ instead. This setup also allows us to run tests in a variety of different setups. For example, it is possible to run the adapter talking to the real machine, our digital twin, or talking to a fully simulated environment. As long as the adapter can talk to the interface, the test can run.

4 Results

4.1 Test outcome and coverage

After setting up our test model and adapter, we were able to run the test on our system with relative ease. The test run executes multiple iterations of configurable length before resetting to the start, to avoid the test runner getting stuck in a neighbourhood of the model it cannot get out of. In the screenshot shown in Figure 6, you can see the different test runs done, and the coverage measurement taken while the test was running:



Number	1	2
Timestamp	14:19:22.976	14:36:21.932
Steps	100	100
Channel	IHBW_Main	IHBW_Main
Label	!RetrieveWorkpiece Completed	?RetrieveEmpty
Direction	←	→
Verdict	Passed	Passed
Tags		

Figure 6: Outcome of a successful test run

Looking at this graph, one obvious question to ask is why the coverage plateaus at 75% and never reaches 100%. Luckily, Axini also visualizes the coverage of the executed test. This visualisation is shown in Figure 7.

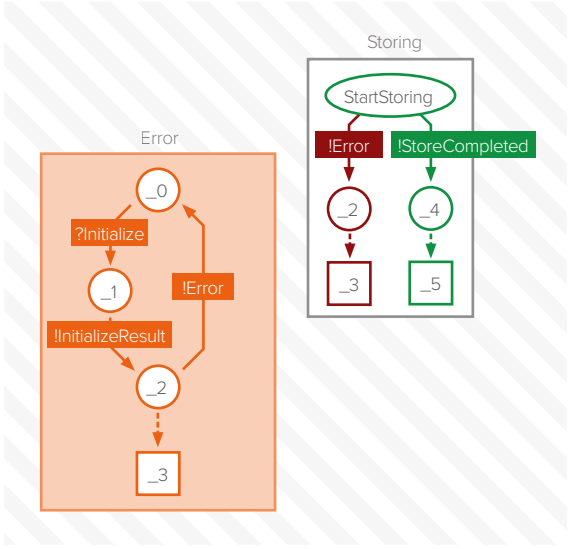


Figure 7: Coverage visualisation of the model

Looking at this visualisation of the coverage, it becomes immediately obvious why 100% coverage is not reached: The system is working well, and the error cases are never seen. Also covering these cases would require some form of error injection, to force the errors.

4.2 Bad weather testing

As you may have noticed, the behaviour model discussed in 3.2 does not contain any errors or not-allowed actions, while there are some error cases visible in the screenshots in the previous section.

This illustrates the fact that while creating the behaviour model, the modeller can decide how much behaviour of the system they want to include. If only good weather behaviour is modelled, the test runner will never decide to try things that are not allowed. Another approach might be to also add all the not-allowed actions to the model, but explicitly adding the expectation that the system refuses this action.

Figure 8 shows part of the model presented in 3.2, but this time with bad weather transitions added in red. Adding these actions tells the test runner it can actually try these actions, but it should expect them to be refused by the system-under-test, without any state change. Running the tester on this model will test many more different state/action combinations than the test before, not only testing all the allowed actions are actually allowed, but also testing that all not-allowed actions are indeed not allowed.

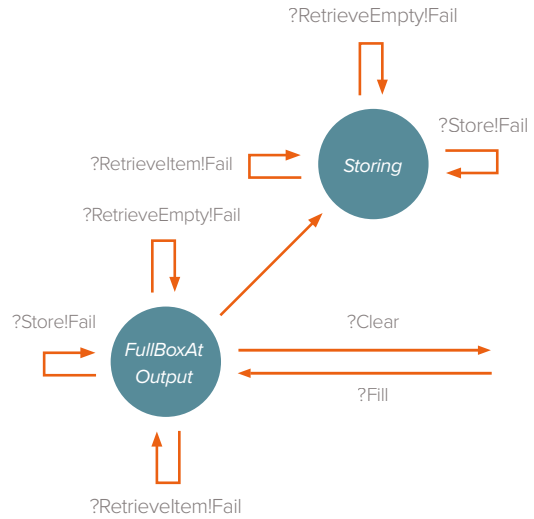


Figure 8: Partial behaviour model with bad weather

4.3 Error injection and the test clamp

Even after adding bad weather into the model, it is possible that the system never fails any of the allowed actions. Since there is code in the system handling failures and recovery, if we cannot force a failure, we cannot reliably cover that code with tests.

To make sure these failures are also covered, we need to add error injection. This means it is no longer enough to have the test adapter connect to the system's provided interfaces, we also need the adapter to influence the system's required interfaces, so simulated failures can be injected, letting us test the failure handling. To be able to set this up, the adapter setup needs to change as shown in Figure 9:

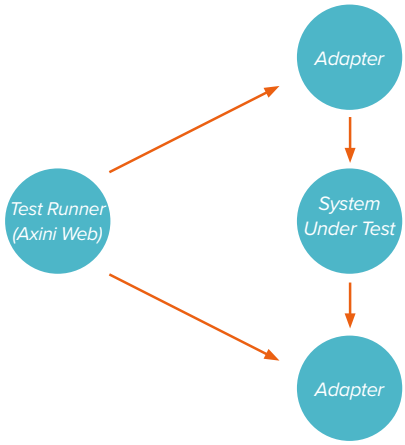


Figure 9: The 'test clamp': Adapters connecting to both the provided and required interfaces of a system

Connecting also to the required interfaces for the system lets the test runner serve as a simulator, including error injection. This setup is very powerful because the test runner can control everything, but it has drawbacks in that it is much more complex to set up, and requires writing and maintaining more models, also for the required interfaces.

For now we have done some initial tests with this setup, but have not settled on a way of setting up our system with the test clamp yet. This is something we plan to investigate further in the future.

4.4 Coverage efficiency

With instrumented executables it is possible to measure the code quality of a test run (model-based or traditional). This means that we can also draw comparisons on the efficiency of the tests, by measuring how many lines of test code or model code are required to reach a certain coverage. Preliminary measurements at one of our customers show that the model-based test approach seems to be about three times more efficient in covering the system's code, compared with traditional tests.

That means that the size of the MBT test models is about three times smaller than the traditional test code, while managing to achieve the same level of code coverage. These are early numbers and more measurements need to be done over time, but this seems to indicate that the modelbased testing approach scales better with scaling system complexity, compared to traditional forms of testing.

5 Conclusion

After using model-based testing on several projects, we have seen a few advantages:

- **Modelling interfaces makes it easier to communicate about behaviour** — Writing down a formal, unambiguous behaviour description has advantages all throughout the software development process: It helps to unambiguously communicate between stakeholders, starting from requirements specification and design, all the way through implementation and quality assurance. We have often seen that having a clear state model can trigger discussions clearing up misconceptions about – even existing – system behaviour.
- **Bugs are found earlier** — We run our modelbased test suite nightly, on the latest development version, effectively starting the QA process before the software is even installed on a real machine. This reduces the number of bugs leaking from development to QA, and therefore also prevents bugs leaking out of QA to customers. As found by NIST[2] (Table 5-1), the cost of software defects scales rapidly the later defects are found in the software life cycle. Therefore finding bugs earlier can save development costs.
- **More coverage, different bugs** — Model-based Testing lives up to the idea that it will find things humans do not think about: It regularly finds issues in scenarios that were not considered beforehand, and that are not covered by other tests.
- **More efficient test code** — We can clearly see that MBT is a more efficient way of increasing test coverage, compared to traditional testing. A much smaller, and easier to maintain test model is able to reach similar test coverage to large traditional test suites. We expect that as the system complexity grows, the difference in effort required to reach higher coverage will only skew more and more in favor of MBT.

On the other hand, MBT is not a silver bullet. There are also some disadvantages and challenges to the approach:

- **Need for experts to get started** — Setting up a new way of testing is not trivial. It requires a different way of working, a different way of integrating into the software ecosystem, and even a different way of thinking/programming. This means that while MBT has the potential to reduce the workload on the QA department, it will require extra effort from the development teams. It is often a good idea to enlist existing expertise holders (like the ICT HTU Center of Excellence) to kick-start a project. Once the foundation is set, the domain knowledge holders can often take over the creation and maintenance of the test model.
- **Not a replacement for traditional tests** — model-based testing should be seen as an addition to existing test suites. Due to the nondeterministic component of the test runner, it is difficult to ensure any specific scenario is tested. If there are some scenarios or user interactions that must absolutely work and should not regress, these scenarios should still be covered by a suite of traditional tests.

6 Recommendations

Overall, we see the value in model-based Testing as an additional tool for quality assurance. If you are building a product with high quality requirements, limited hardware time or limited QA availability, we believe the addition of MBT can make a valuable contribution to the quality of your product. Getting started on your own might be hard, so we recommend getting in touch with us or other knowledge holders in this field, to discuss if this technology can help you, and how you can kick-start your journey into model-based testing.

References

[1] Axini: Model-Based Testing platform. https://www.axini.com/en/.

[2] N. I. of Standards & Technology. The economic impacts of inadequate infrastructure for software testing. Technical report, NIST, 2002.

Author: Perry van Wesel - Designer ICT High Tech

ICT High Tech
Center of Excellence
centerofexcellence@ict.nl
+31 (0)88 908 2000

ICT High Tech