Generator-based Testing: A State by State Approach

Chris Struble

cstruble@meteorcomm.com

Abstract

Learning a new software testing technique and applying it successfully in your career is deeply satisfying. What if your organization is heavily invested in a different testing technique? How can you convince your colleagues to try your technique, and help them to adopt it successfully?

Many organizations use Example-based testing (EBT), where test cases are manually designed one example at a time, then executed automatically in a repeatable way. EBT techniques such as Test-Driven Development (TDD), Behavior-Driven Development (BDD), and Acceptance Testing have been widely adopted. EBT falls short because it cannot be used to test examples that no one thought of.

EBT shortcomings had led to development of techniques for automated generation of test cases. In Generator-based Testing (GBT), software behavior is described to a tool that generates test cases in a non-deterministic way, which are then executed automatically. Three examples of GBT techniques are Model-based testing (MBT), Property-based testing (PBT), and Fuzzing. Each of these techniques have found high profile defects that escaped EBT testing. Even so there has not been widespread adoption of these techniques.

This paper will present an approach to doing GBT that makes it more accessible, and easier to progress through states of adoption. The approach is based on more than 20 years of experience using MBT. The paper will also describe an MBT framework developed in Ruby using open source tools in the past year, and the results of introducing it in a workplace where EBT techniques were already widely adopted.

Biography

Chris Struble is a Senior Software Developer in Test (SDET) at Meteorcomm LLC in Renton, Washington. He has 25 years of experience as a software professional, primarily in quality and test automation, with excursions into development, agile team leadership, and continuous delivery. He holds an MS in computer science from Walden University and an MS in mechanical engineering from the University of Houston. In his spare time, he writes music and fantasy fiction, and plays guitar and computer strategy games. He lives in Renton, Washington with his wife, daughter, and two cats. Find him at: chrisstrublewrites.blogspot.com

1 Introduction

When I started my software engineering career in the early 1990s, most software testing was a manual process of trying out a sequence of steps, writing them down, and executing them again.

In 2020, automated test execution is widely adopted, with test execution frameworks available in many programming languages, and for every level of testing, from unit testing to system testing, and for functional and non-functional aspects of software.

Automated test execution transformed the job roles of software developers and software testers, moving them closer together than ever before. Today testers must be able to write code, and developers must be able to write tests. In some workplaces the distinction has disappeared altogether. For this paper "tester" is used to mean anyone doing software testing, regardless of job role.

For most testers, test case design remains a manual process of writing sequences of steps one example at a time. This practice is so common that conventions have been developed to make it easier.

Here is an example written in the Gherkin syntax of the Cucumber test framework:

```
Given I have a locomotive travelling at 80-mph
When it approaches a curve with a 30-mph speed limit
Then an overspeed warning message should be sent to the operator console
And the operator does not slow the train within 30 seconds
And I should see the locomotive slow safely to a stop automatically
```

Most testers would immediately recognize the intent of this example, because people naturally communicate and learn by example. Examples are a form of storytelling, and stories needs concrete characters who do concrete things, even when some of the characters are systems and software.

The above example describes a scenario that can occur with Positive Train Control (PTC), an automatic safety system for railroads in the United States. The functionality in this example is designed to prevent an overspeed derailment, such as the one that occurred on December 18, 2017 near DuPont, Washington (McNamara, 2019).

The above example is concrete, with specific speeds and time intervals. Ideally, we would like to make a general claim about software behavior, more like this:

```
Given I have a locomotive travelling at some speed
When it approaches a curve with a lower speed limit
Then an overspeed warning message should be sent to the operator console
And the operator does not slow the train in time
And I should see the locomotive slow safely to a stop automatically
```

David MacIver (2019) describes test cases that "use a concrete scenario to suggest a general claim about the system's behavior" as "example-based tests".

In this paper Example-based Testing (EBT) is defined to mean any practice where:

- Test cases are manually designed one example at a time
- With the goal of suggesting that the software works
- By demonstrating automatically that each example works

EBT practices including Test-Driven Development (TDD), Behavior-Driven Development (BDD) (such as the Cucumber examples above), and Acceptance Test-Driven Development (ATDD) are widely used for software testing today.

EBT has inherent shortcomings. A single concrete example can only suggest that a general claim is true. It takes many concrete examples to have confidence in a general claim. People, including most testers, are not very good at thinking of a thorough set of examples.

A consequence is that defects escape the software development process and are first seen by users after the software is released. Some escaped defects occur because of schedule and cost constraints of software projects, but many escaped defects are examples that no one even thought of.

What if we could automate test case design, using a computer program to “think of” more examples?

That is the goal of Generator-based Testing (GBT), defined as any software testing practice where:

- Test cases are generated automatically from a description of the software behavior
- With the goal of discovering defects in the software
- By demonstrating automatically if each generated test case works, or not

This paper will focus on three practices that fit this definition: Model-based testing (MBT), Property-based testing (PBT), and Fuzzing. Each of these practices have been in use for more than 20 years, enough time to have found many important escaped defects. Here are just three real-world examples:

- Mars Polar Lander crash, 1999
    - The Mars Polar Lander was on lost December 3, 1999, just before touchdown on Mars. After the crash, a NASA team used T-VEC, an MBT test generation tool, to successfully identify the likely cause of the crash. A fault in the Touchdown Monitor system caused the lander to shut down the descent engines prior to reaching the surface (Blackburn, 2002).
- HeartBleed OpenSSL Vulnerability, 2014
    - HeartBleed was a security vulnerability in the OpenSSL cryptographic library. It was independently discovered in April 2014 by Neel Mehta of Google, and a team at Codenomicon (now Synopsis) using a Fuzzing test tool. The vulnerability was undetected for two years. (Synopsis, 2014)
- Volvo AUTOSAR Emergency Braking System Fault. 2015
    - Volvo hired Quviq to perform acceptance testing of the AUTOSAR software on its vehicles. Quviq used its QuickCheck PBT tool to detect over 200 problems, including a serious problem where the emergency braking system could be deprioritized over non-critical functions, such as adjusting the volume. (Hughes, 2015)

Despite successes like these, GBT practices are rarely used. A global job site had over a thousand job listings mentioning EBT practices in the qualifications, while GBT practices were mentioned in only five.

For GBT practices to become mainstream, I believe they must be presented as complementary practices that make up for EBTs shortcomings. EBT and GBT can and should coexist in a tester’s toolkit, and in the same software development group. The following GBT framework attempts to support that idea.

2 Generator-based Testing (GBT)

To practice GBT requires artifacts and tools that work together. In order of appearance, these are:

- A Description of the software behavior the Tester wants to test
- A Generator that generates Test Cases from the Description
- An Executor that executes each Test Case against the software
- An Oracle that determines if each Test Case passed or failed
- A Repeater that tells the Executor to rerun each failed Test Case
- A Shrinker that finds a minimal Repro Case that reproduces each failure

Figure 1 shows a workflow of how a tester would use these artifacts and tools together.

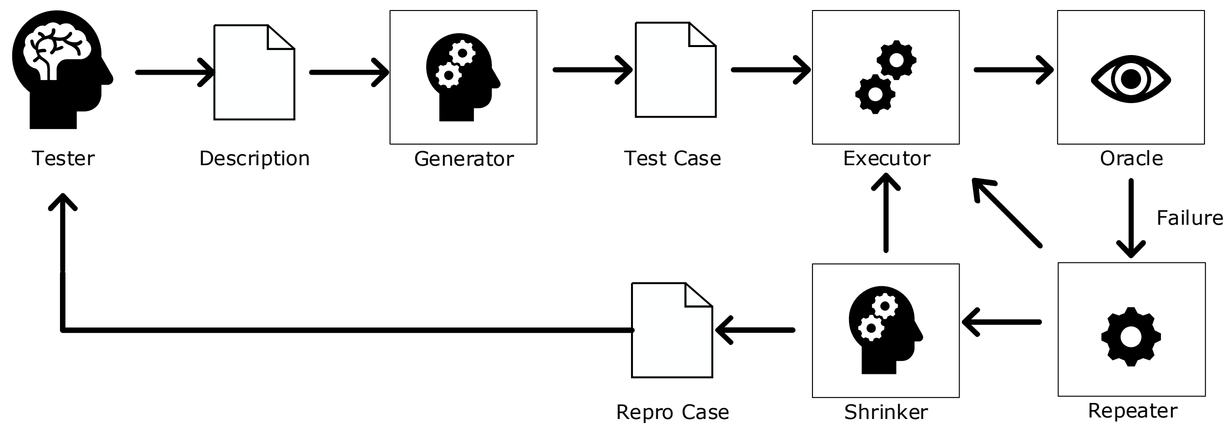

Figure 1 – Process of Generator-based Testing (GBT)

A Description is an artifact such as a file, graph, or source code that formally describes a behavior to be tested. The Description is created by the Tester.

A Generator is a tool with source code tightly coupled to the format of the Description. It processes the Description and outputs Test Cases.

Generators are non-deterministic. Each time a Generator processes the same Description, it may produce a different set of test cases. This is because Generators use search algorithms or random algorithms to generate test cases. This is a strength in GBT since the goal is to find defects.

Executors and Oracles should already be familiar from automated test execution. An Executor runs each test step against the software under test, and an Oracle determines if that step passed or failed.

A Repeater is a tool that reruns the Executor for Test Cases that were failed by the Oracle. If a Test Case fails again, it is considered a defect candidate and sent to the Shrinker.

A Shrinker is a tool that uses a search algorithm to reduce a test case that consistently fails, into a shorter Reproducible (or Repro) Case that reproduces the same failure.

Shrinkers are necessary because Generators can produce very long test cases, with many steps, only some of which may be important to the failure that occurred.

The Repro Case is then analyzed by the Tester to determine the cause of the failure. The root cause might be an error in the Description, Generator, Executor, Oracle, or a mismatch of one of these with the behavior of the software under test. The appropriate action is taken, and the process is repeated.

Any of the steps in the workflow can be done manually. The greatest benefits occur when each step of the workflow is automated, and when the output of one step flows automatically into the next step.

To make these concepts more concrete, the next section will discuss the practice of Model-based Testing (MBT) in more detail, along with a case study.

3 Model-based Testing (MBT)

Model-based Testing (MBT) is one type of GBT. In MBT, the Description is a “model”, such as a finite state machine, UML state chart, process flow diagram, or similar directed graph. The model describes actions that cause changes in the state of the software under test.

In MBT, representing the model visually as a graph is essential. A model graph can be used as a specification, to communicate the tester’s understanding of the software behavior to. This requires a Model Editor, a tool to display the graph, edit it, and save it back to a file readable by the Generator.

MBT has been practiced since the 1990s or earlier. Some milestones include:

- First paper using the term “MBT” (Apfelbaum, Larry and Doyle, John. 1997)
- First dedicated conference workshop (A-MOST. 2005)
- First textbook (Utting, Mark and Legeard, Bruno. 2007)
- First certification (ISTQB. 2015)

My first experience with MBT was in 2000, when I used TestMaster, a commercial MBT tool, to generate test cases for a laser printer driver installer, finding over 100 new defects in a six-month project (Struble, Chris. 2004). That project made a lasting impression on me.

I found model graphs to be a natural and effective way to generate test cases, and I never stopped using them, even when automated generators were not available to me. I continued to try new MBT tools periodically or wrote my own.

Here are a few more highlights from my use of MBT in my software testing career:

- I released Hanno, an open source MBT web testing framework in Java (Struble, Chris. 2008)
- I tested .NET web applications using Microsoft Spec Explorer, in 2012
- I evaluated Conformiq Creator, a commercial MBT tool, in 2016

My efforts were successful in the sense that I learned from them, raised awareness about MBT among my colleagues, and in several instances successfully found product defects. None of these efforts led to MBT to become an ongoing practice at the organizations I worked for.

One impediment to software teams adopting MBT is that the mental shift required to make the transition to MBT takes several months. For teams to adopt it, several people need to go through the learning process at the same time, or there needs to be sustained support for it. Few employers are willing to make that kind of investment in tester training.

Once mastered, MBT can be quickly reapplied years later. One tester used MBT early in his career and returned to it 20 years later (Lowry, Benjamin. 2020).

It is easier to get started with MBT now than when I started. Several well supported open source MBT tools exist today (Micskei, Zoltán. 2018). This was helpful when I tried MBT again in 2019.

4 Meteorcomm MBT Case Study

In 2019 I joined Meteorcomm LLC (2020), a telecommunications company headquartered in Renton, Washington.

Meteorcomm develops a messaging system for railroads called Interoperable Train Control Messaging (ITCM). ITCM maintains communication between locomotives, waysides and railroad back-offices via radios and IP networks. Figure 2 shows the major components of the ITCM system.

The main application of ITCM is Positive Train Control (PTC), a safety system that was mandated by the Rail Safety Improvement Act of 2008. PTC is designed to stop a locomotive if it is not able to continuously prove to the back-office that it is safe for it to proceed. Over 90% of the track miles that implement PTC in the United States use the Meteorcomm system.

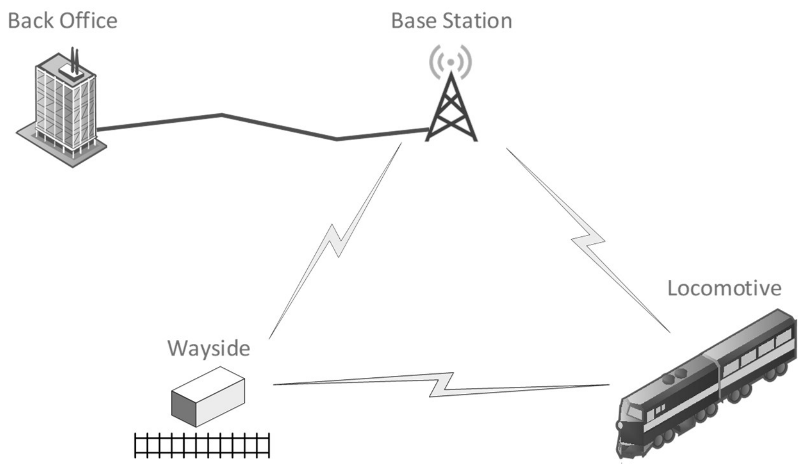

Figure 2 – Interoperable Train Control Messaging (ITCM) System

Meteorcomm is rearchitecting its back-office systems as Docker containers running as “pods” in a Red Hat OpenShift (Kubernetes) cluster. A library of several thousand functional tests, written in the Ruby programming language for the Cucumber BDD test framework, are being rewritten for the new architecture. Sequence diagrams are used extensively to model message flow through the system.

Several times a year development teams get two-week “innovation sprints” to work on projects of our choosing. I used one of those intervals to introduce MBT to Meteorcomm. I wanted an MBT tool in Ruby so it would work with our test code, but I could not find one, and writing one from scratch would have taken too much time.

Instead I used an open source MBT tool called GraphWalker (Karl, Kristian. 2020). GraphWalker has been developed since 2005 and is the most fully featured free MBT tool. It is written in Java but can be controlled with any programming language through a REST interface.

In two weeks, I wrote a Ruby program called Test Generator to generate and run tests for ITCM using GraphWalker. Here is how it works, using the terms in the GBT framework:

- Description: A finite state machine model is created in GraphWalker JSON format.
- Generator: GraphWalker running as a REST service.
- Executor and Oracle: A Ruby model class with methods for each element in the JSON model file. Edge (transition) methods execute actions, and vertex (state) methods verify that the software is in the correct state.
- Test Case: The sequence of steps generated by GraphWalker can be saved to a “walk file”.
- Repeater: Test Generator can re-run a walk file, without using GraphWalker to re-generate.
- Shrinker: Currently a manual process. To be automated in the future.

The overall workflow can be summarized as follows: Test Generator loads the JSON model into GraphWalker, then GraphWalker traverses the model one element at a time. For each element, Test Generator calls the Ruby model class, until GraphWalker reaches 100% edge (transition) coverage.

Figure 3 shows a simple GraphWalker model of AMQP Sender (called "tx" in the model), rendered using the AltWalker Model Editor (Altom. 2020). AMQP Sender is a test pod that runs in the OpenShift cluster. It provides a REST interface to send messages using the Advanced Message Queuing Protocol, simulating messages sent by locomotives or waysides.

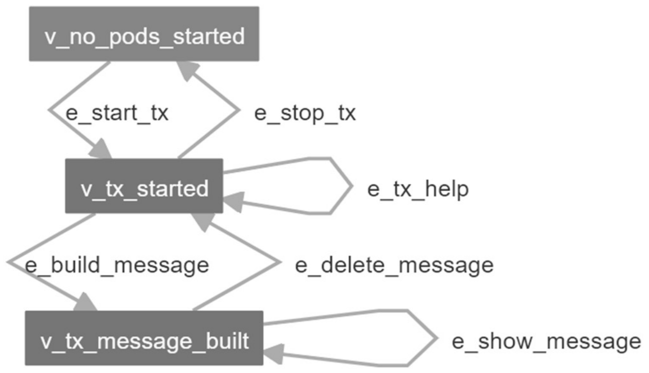

Figure 3 – GraphWalker model of the AMQP Sender test pod

This model has three vertices and six edges. The initial vertex "v_no_pods_started" means the pod is not started. Once the pod is started, it can be stopped, display a help page or build a message. Once a message is built, the message can be displayed or deleted.

GraphWalker typically took 12-16 steps to reach edge coverage using Djikstra's algorithm.

I presented the Test Generator and showed it generating and running tests for the AMQP Sender model, calling the backend test library used by our existing Cucumber test cases. I got positive feedback that encouraged me to keep working on it.

I spent another two weeks enhancing Test Generator with more features, improved documentation, and unit tests. I built a more complex example model. This model (not shown) has three pods: AMQP Sender, MRG3 Broker, and AMQP Receiver. AMQP Sender sends a message to the broker, which stores the message in a queue, until it is read by AMQP Receiver. The model had 10 vertices and 28 edges. GraphWalker typically took 84-100 steps to reach edge coverage on this model using Djikstra's algorithm.

Finally, I set up a half-day workshop with the testers in my department. Six people attended and got to use Test Generator to model and generate tests in a group exercise. None of them had used MBT before. I asked in a survey if they would consider using MBT in their day-to-day work. Here are some responses.

"Learning new approach to testing. Random generation of tests, down different possible paths every time. I felt this approach would be useful for generating every corner case which otherwise wouldn't get done during normal test automation."

"I think this type of testing could be very beneficial if you get the vertices and edges to match up well to parts of the code base. This may require some trial and error. I would be interested in knowing if there's any sure procedure for converting design documents to these graphs."

"I need to think about whether this really adds value in our testing domain. While I see immediate value and application in web/UI testing, I haven't seen adoption of this technique in the back-end systems and integration testing domains I've experienced throughout my career."

After these efforts, Test Generator was added to the list of officially allowed testing frameworks at Meteorcomm. I hope to use it to search for defects in ITCM in the coming year.

5 Property-based Testing (PBT) and Fuzzing

In 2019 I became aware that MBT isn't the only game in town. In a web search I found a video by John Hughes called "Don't Write Tests." He does another form of generative testing called Property-based Testing (PBT).

PBT started in 1999, when Hughes co-wrote a Haskell library called QuickCheck for generating test cases for functional programming languages (Hughes, 2015). Assertions are made on "properties" that functions must satisfy for all inputs, and many examples are randomly generated attempting to falsify the assertion. If a falsifying example is found, QuickCheck automatically tries to shrink the test case to find a minimal test case that reproduces the same failure.

PBT has been described as "a more structured approach to fuzzing". Fuzzing is a generative testing practice where random inputs are entered into a computer program until it crashes. Fuzzing has been used since the 1980s, and today is often used to find new security vulnerabilities, such as HeartBleed (Synopsis, 2014), or to test any software where random crashing bugs are a concern.

PBT is not limited to functional programming languages. QuickCheck has been ported to 35 languages. Hypothesis, a Python library for PBT, was introduced in 2015 (MacIver, David. 2019). It has so far been ported to Java and Ruby. Over 2000 open source projects are already using Hypothesis.

PBT is not limited to unit testing. It is used for integration testing and system testing, and even testing of stateful systems (using what Hughes calls "models", though these are not the models of MBT).

Learning about PBT and Fuzzing made me realize I needed to expand my concept of GBT to encompass them. The "shrinker" concept is used by both, and the term "example-based testing" comes from PBT.

I haven't use PBT yet. I hope to try it in the coming year.

6 What MBT and EBT could learn from PBT

Looking at PBT from the perspective of an MBT expert, I believe that MBT could benefit from some of the concepts that PBT has implemented, in particular:

- Model in code: Most testers would prefer to use source code rather than graphs to capture their understanding of test behavior. This avoids context switching between graphs and code which can slow down the test generation process.
- Automate shrinking: MBT generates long sequences of test steps, especially when tests are generated and run at the same time. Most MBT tools do not implement shrinking, leaving it for the tester to do manually. Automated shrinking would make MBT a more efficient workflow.
- Embrace randomness: Randomness is a strength when searching for defects. PBT embraces this. MBT's search algorithms are also non-deterministic, but MBT tool vendors don't always emphasize this. This is misguided. MBT should embrace randomness as a selling point.

EBT could also benefit from randomness. Running manually designed EBT test cases in random order is an easy way to detect test cases in EBT test suites that interfere with each other and remove flakiness.

7 A State by State Approach to Test Learning

Mastering a new software testing practice takes time. Embedding it in a software development group takes even longer and involves passing through a series of states of personal and organization learning. What follows is a model, in GraphWalker format, of such a learning process.

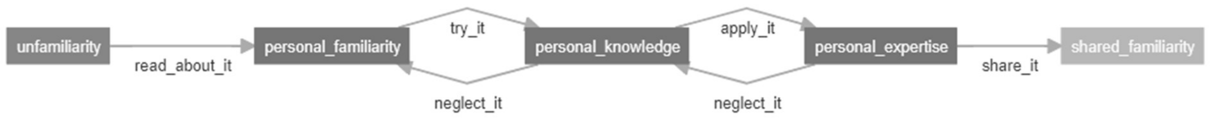

Figure 4 – Personal Learning States

Figure 4 shows the states of personal learning. In the beginning, the tester is unfamiliar with a new testing practice. If they read about it, they become familiar with it. If they try it, perhaps installing tools and working through examples, they achieve personal knowledge. Then they have a choice. They can try to become an expert, by applying it in a real project. This might take a few months. If that isn't possible right then, they can set it aside. After a year or so, neglect sets in. Then to get back to the state of personal knowledge, they would have to try it again.

Once they become an expert, they have another choice. Share the expertise with their colleagues, in a presentation or demo, or neglect it. If they neglect it, they can always apply it again later to get back to be an expert. If they share the expertise, then the tester has reached a state I call "shared familiarity". They are an expert, and their colleagues know they are an expert.

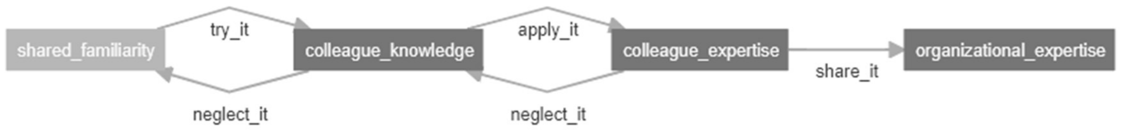

Figure 5 – Organization Learning States

Once the expertise has been shared, the organization then can learn. This process is shown in Figure 5. Each of the tester's colleagues has the choice to try the new technique, or not. The original tester can help by leading a workshop where colleagues get to try it as a group. For a colleague to become an expert, they must apply the knowledge on a real project. Once the organization has at least two experts in a testing technique, they should be able to maintain that expertise in the organization.

Organization learning of a new testing practice can regress from neglect at any point. Ongoing support from management is essential. Dedicated innovation weeks, training weeks, brown bags, internal conferences, and other opportunities for knowledge sharing on a regular basis, encourage testers to learn new practices and to maintain and spread that knowledge in the software development group.

Summary

This paper gave a definition of Example-based Testing (EBT), and a definition of Generator-based Testing (GBT) that encompasses Model-based Testing (MBT), Property-based Testing (PBT), and Fuzzing. It described three major escaped defects that were found with each of these practices. It described the tools needed to practice GBT. It discussed a case study of introducing MBT to a software development group by creating an MBT framework in the same programming language as existing EBT test libraries. Finally, it presented a state machine model for personal and organizational learning, that may help in adopting GBT practices (or any new testing practices) in a software development group.

Acknowledgements

I want to thank managers past and present who encouraged my explorations: Irv Tyrrell, Denzil Dwelle, Rita McCann, and Starr Parker. Thanks to Harry Robinson for inspiration and encouragement. And most of all, thanks to my wife LeAnne Struble for her patience and support.

References

McNamara, Neal. 2019. NTSB Issues Final Report On DuPont Amtrak Derailment. https://patch.com/washington/seattle/ntsb-issues-final-report-dupont-amtrak-derailment (accessed 7/14/2020)

MacIver, David. 2019. “In Praise of property-based testing”, Increment.com, Issue 10, August 2019, https://increment.com/testing/in-praise-of-property-based-testing/ (accessed 06/28/2020).

Blackburn, M.R. 2002. Mars Polar Lander fault identification using model-based testing https://www.researchgate.net/publication/4004301_Mars_Polar_Lander_fault_identification_using_model-based_testing (accessed 06/28/2020)

Hughes, John. 2014. Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane https://www.cs.tufts.edu/~nr/cs257/archive/john-hughes/quviq-testing.pdf (accessed 6/28/2020)

Synopsis. 2014. https://heartbleed.com/ (accessed 6/28/2020)

Apfelbaum, Larry and Doyle, John. 1997. “Model Based Testing”, Software Quality Week Conference, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.1342 (accessed 06/28/2020)

A-MOST. 2005. Advances in Model-Based Software Testing https://conf.researchr.org/series/a-most (accessed 06/28/2020)

Utting, Mark and Legeard, Bruno. 2007. *Practical Model-Based Testing: A Tools Approach*. San Francisco: Morgan Kaffmann, ISBN 978-0-12-372501-1.

International Software Testing Qualifications Board, Foundation Level Model-Based Tester Certification. 2015. https://www.istqb.org/certification-path-root/model-based-tester.html (accessed 06/28/2020)

Struble, Chris. 2004. Model-Based Testing of Installers in a Development Environment. http://testoptimal.com/ref/MBT_Installers_Dev_Env.pdf (accessed 06/28/2020)

Struble, Chris. 2008. Hanno Model Based Web Testing Framework. https://sourceforge.net/projects/hanno/ (accessed 06/28/2020)

Lowry, Benjamin, 2020. Model based-testing: Gone and back again https://speakerdeck.com/bplowry/model-based-testing-gone-and-back-again-net-perth-march-2020 (accessed 06/28/2020)

Micskei, Zoltán. 2018. Model-based testing (MBT). http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html (accessed 06/28/2020)

Meteorcomm LLC. 2020. https://www.meteorcomm.com/ (accessed 06/28/2020)

Karl, Kristian. 2020. GraphWalker. http://graphwalker.github.io/ (accessed 06/28/2020)

Altom Consulting. 2020. AltWalker Model Editor https://altom.gitlab.io/altwalker/model-editor/#/visual-editor (accessed 06/28/2020)