Faculty of Engineering of the University of Porto

An overview of testing automation techniques

Pedro Tavares

November 15, 2017

Contents

1 Introduction

Software must be tested to have confidence that it will work as it should in its intended environment. Software testing needs to be effective at finding any defects which are there, but it should also be efficient, performing the tests as quickly and cheaply as possible. Software testing can provide an objective, independent view of the software quality and performance that allows the business stakeholders to understand the quality of software implementation. It has been considered as a time consuming activity that calls for enhanced and powerful test techniques with the intent of finding software bugs but automating software testing can significantly reduce the effort required for adequate testing, and significantly reduce the time needed to run tests that would take hours to run manually.

Automating tests is a skill, but a very different skill from testing. Many organizations find that it is more expensive to automate a test than to perform it once manually. In order to gain benefits from test automation, the tests to be automated need to be carefully selected and implemented. Automating a test affects how economic and evolvable it is. Once implemented, the cost of running an automated test is much more cheaper than perform it manually. However, automated tests generally cost more to create and maintain. The better the approach to automating tests, the cheaper it will be to implement them in the long term.

There are a number of ways in which testing tools can automate parts of test case design. These tools are sometimes called test input generation tools and their approach is useful in some contexts, but they will never completely replace the intellectual testing activities. One problem with all test case design approaches is that the tool may generate a very large number of tests. Some tools include ways of minimizing the tests generated against the specified criteria. However the tool may still generate far too many tests to be run in a reasonable time. The tool cannot distinguish which tests are the most important, since it requires creative intelligence only available from humans. Test generation tools rely on algorithms to generate the tests. The tools will be more thorough and more accurate than a human tester using the same algorithm, so this is an advantage. However, a human being will think of additional tests to try, may identify aspects or requirements that are missing, may be able to identify where the specification is incorrect based on personal knowledge and may be able to determine which are the most important test cases to run. The best use of test generation tool is when the scope of what can and cannot be done by them is fully understood.

In this paper, we will explore and compare some of the techniques and strategies used to take full advantage of testing automation.

2 Data-Driven Testing

Data-driven testing is a scripting technique that stores test inputs and expected outcomes as data, normally in a tabular format, so that a single driver script can execute all of the designed test cases [1]. Usually, testing scripts that don't follow a data-driven approach, have data attached into them. When test data needs to be updated the actual test script must be changed as well. This might not be a big deal for the person who originally implemented the script but it may become a problem to a test engineer that has less programming experience.

Data represents a large part of system and test specification. By considering how data is defined and used during testing we can increase the opportunity of reuse as well as reduce future maintenance costs for subsequent releases of the system being developed. That's what make data-driven testing extremely useful since one single test script can be used to run different tests, reducing the number of the overall test scripts needed to implement all the test cases.

A suitable scenario where data-driven testing can be applied is when two or more test cases requires the same instructions but different inputs and different expected outcomes. The refactoring would result in one single test script and a shared input data file.

2.1 Test Data

Test Case	Input 1	Operation	Input 2	Result
Addition 01	5	+	5	10
Addition 02	5	+	10	15
Subtraction 01	10	-	5	5
Subtraction 02	10	-	0	10
Multiplication 01	5	*	5	25
Multiplication 02	5	*	0	0
Division 01	10	/	2	5
Division 02	10	/	10	0

Table 1: Data-driven test data with mathematical operations.

2.2 How to store test data?

Data-driven testing calls for tabular data and the first instinct is to use spreadsheet programs to edit it [2]. Spreadsheet files like comma-separated-values (CSV) are handy because they are very easy to parse but unfortunately the data becomes inconsistent when edited by two or more different spreadsheet programs. But the biggest problem with storing test data into any kind of flat file is scalability [2]. If for example test data is created and edited in several workstations and used in multiple test environments it gets hard to have the same version of the data everywhere. Moving the data files to a central database, backed up by a web-based interface in order to manage the test data, is a suitable solution when scalability is a requirement. This way, the driver scripts can read test data directly from the database instead of parsing CSV files before running the test suite.

2.3 Processing Test Data

Implementing a script for parsing data-driven test data can be surprisingly easy either using data files or databases [2]. Data is processed line by line and split into cells that represent the test inputs making the data available to be used in the test scripts. The main drawback is that the parser will always be limited in extensibility and it will crash if the test data format changes. Whatever design decisions are made about the driver script, the test data must reflect to it [1]. This means that if we change either the driver or the format of the data, we need to make sure they are always in sync.

Listing 1: Example of data-driven driver script.

```
const TestData = require('./testData');
const Calculator = require('./calculator');

TestData.testCases().forEach(function(testCase) {
  const input1 = testCase.input1();
  const operation = testCase.operation();
  const input2 = testCase.input2();
  const expected = testCase.result();

  execute(input1, operation, input2, expected);
});
```

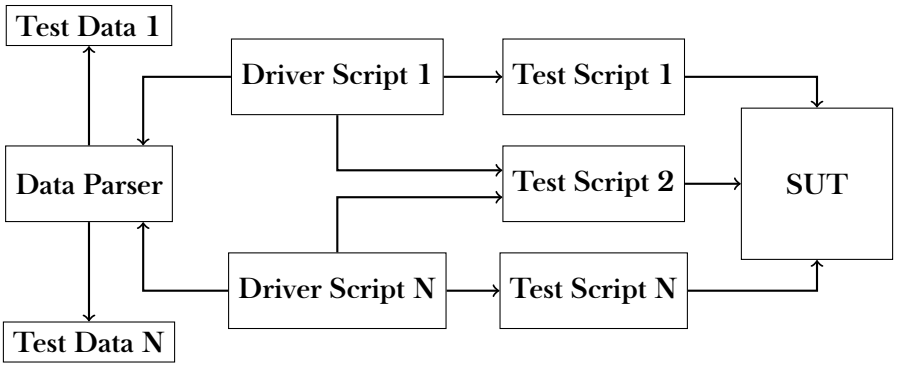

Figure 1: Data-driven system design approach.

In this data-driven approach there are a few elements that are crucial and must be implemented:

- **Data Parser**: It is responsible for parsing the test data, either from a CSV file or a database table. Its main task is to process test data and return it to the driver script providing an easy and neutral access to the test data.
- **Driver Script**: It is responsible for driving the test execution process using the testing functionality provided by the test libraries, that interact with the system under test, and the test data, that is read by the data parser.

2.4 Advantages of data-driven testing

A major advantage of the data-driven testing is that the format of the data file can be tailored to suit the testers needs [1]. The data file can contain comments that the script will ignore but that will make the data file much more understandable and, therefore, maintainable even by non-technical people. The easier it is, the quicker and less error prone it will be.

With data-driven testing we can really start to benefit from test automation [1]. It is possible to implement many more test cases with very little extra effort since all we have to do is specify a new set of input data and expected results for each additional test case.

2.5 Disadvantages of data-driven testing

The biggest limitation of the data-driven approach is that all test cases are similar and creating new kinds of tests requires implementing new driver scripts that understand different test

data. Thus the test data and driver scripts are so strongly related that changing either requires changing the other. In general, test data and driver scripts are strongly coupled and need to be always synchronized [1]. If we look closer at the input data presented in Table 1 we can see that was designed only to accept two inputs and in order to support operations like 10 * 5 - 5 = 45 would require major changes on both test data and driver scripts.

The initial set-up effort of a data-driven testing framework is high [1]. Writing the right drivers need to be done by someone with programming skills, and not all of teams have resources to spend time on setting up testing frameworks. Although, the benefits gained will vastly outweigh this upfront cost when the test suit grows.

It is not appropriate for small systems where the cost of writing test cases is not so high [1]. If a data-driven approach is used it will entail more work, and will seem excessive. On the other hand, large and complex systems gain far more in saved effort than you put in.

3 Keyword-Driven Testing

Keyword-driven testing is a logical extension to data-driven testing [1]. Besides test data, it also makes use of keywords in order to instruct how the data is read from the external data source and to be interpreted by the test scripts. It takes the concept of data-driven testing even further [2] by adding keywords driving the test executing into the test data with the desire to be able to specify automated test cases without having to specify all the excessive detail. The test data file is expanded and it becomes a description of the test case using a set of keywords to indicate the tasks to be performed. Obviously, the driver script has to be able to interpret the keywords in order to execute the desired test case.

Back to the previous scenario on data-driven testing where the operation 10 * 5 - 5 = 45 would require major changes to both test data and driver scripts. The keyword-driven testing is the perfect technique to perform that kind of scenarios. They could be extended with no further development since the keywords are already implemented by the driver script. This way, the driver script is no longer tied to a particular feature of the software under test, nor indeed to a particular application or system [1].

This technique takes a descriptive approach to test case implementation since the test file describes the test case, by stating what the test case does and not how it does it. To implement an automated test case we have only to provide a description of the test case

more or less as we would do for a knowledgeable human tester. By using this approach we can build knowledge of the system under test into our test automation environment. People with business knowledge can concentrate on the test files while people with technical skills can concentrate on the driver scripts since it is possible to develop the test cases separately from the test scripts due to the abstraction created by the test files.

3.1 Test data

Test Case	Keyword	Value
Addition 01		
	Operand	5
	Operator	+
	Operand	5
	Result	10
Addition 02		
	Operand	5
	Operator	+
	Operand	10
	Operator	-
	Operand	1
	Result	14

Table 2: Keyword-driven test data with mathematical operations.

There is no real difference between handling keyword-driven and data-driven test data [1, 2], but we need to make a big decision when designing a keyword-driven framework: the abstraction level of the keywords to be used [2]. When testing higher level functionality like business logic, low level keywords tend to make test cases very long, defined on Table 2, while higher level keywords, defined on Table 3 are much more usable. These higher level keywords shouldn't be implemented directly into the framework. Instead, they should be implemented using the same technique used for designing test cases. The main benefit in making it possible to create new keywords using the test design system is that they can be created and maintained easily without any programming skills. The drawback [2] from this approach is that the driver script will get more complex since it has to handle two kinds of keywords.

Test Case	Keyword	Value
Addition 01		
	Input	5
	Add	5
	Result	10
Addition 02		
	Input	5
	Add	10
	Subtract	1
	Result	14

Table 3: Keyword-driven test data with higher level keywords.

3.2 Processing test data

The driver script should be very similar to the one presented in Listing 1 and must be able to interpret the defined keywords and execute the matching action using the assigned values. Generally the number of scripts required for this approach is a function of the size of the software under test rather than the number of tests [1].

Listing 2: Example of keyword-driven driver script.

```
const TestData = require('./testData');
const Calculator = require('./calculator');

TestData.testCases().forEach(function(testCase) {
  testCase.keywords().forEach(function(keyword, value) {
    execute(keyword, value);
  });
});
```

In this keyword-driven approach there are a few elements that are crucial and must be implemented:

- **Data Parser**: Share the same responsibility as the data-driven parser defined on Figure 4, but has to handle multiple instructions per test case (Input, Add, Result).
- **Driver Script**: Share the same responsibility as the data-driven driver defined on Fig-

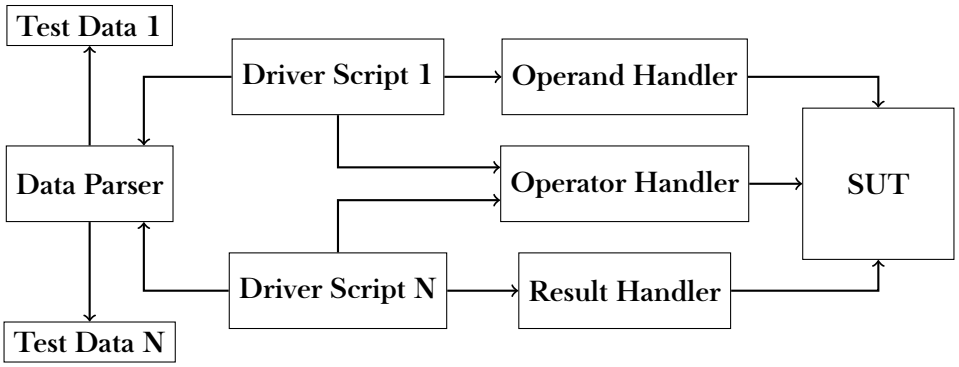

Figure 2: Keyword-driven system design approach.

ure 4, but instead of dealing directly with the test libraries, it needs to use specific handlers to translate the keywords and the data values that are read by the data parser in order to execute the tests.

- **Handlers**: Each action keyword corresponds to a single fragment of a test script (Operator, Operand, and Result handler), allowing the test execution tool to translate a sequence of keywords and data values into executable tests.

3.3 Advantages of keyword-driven testing

Keyword-driven testing has all the same benefits as data-driven testing since it shares the same principle of making easier to create new kinds of tests, even for non-programmers. But despite their resemblances, keyword-driven testing is a big step forward from pure data-driven testing where all tests are similar and creating new tests always require new code in the framework. Once the basic driver scripts are in place, it is possible to implement thousands of test cases with only a few hundred test scripts. This approach not only speeds up the implementation of automated tests, but also makes it possible to use testers without programming skills to implement them. The beauty of this approach is that it can be implemented in a tool independent way so that we can re-implement the driver scripts in another testing tool or programming language without loosing the investment in the test cases. All of the information about what to test is now contained in the data table, so the separation of test information from implementation is the greatest advantage in this technique.

3.4 Disadvantages of keyword-driven testing

The main problem with the keyword-driven approach is that test cases tend to get longer and more complex than when using the data-driven approach. If we look at the example from Table 1, the test case Addition 01 has only one line and it's very restrictive in terms of inputs. On the other hand, the same test case on Table 2 has four lines, is more complex to implement, but has lot more flexibility.

4 Hybrid-Driven

5 Model-Based Testing

Model-based testing is a software testing technique that uses test generation algorithms and a behavioral model of the system under test to generate test cases [3]. Unlike traditional development techniques which tend to focus on implementation, model-driven software development stresses the use of models at all levels of the software development process [4]. Like the data-driven and keyword-driven approaches to testing, model-based testing relies heavily on abstraction. However, the definition of a model varies greatly, depending on the approach [5]. Instead of writing hundreds of test cases the test designer writes an abstract model of the system under test, and then the model-based testing tool generates a set of test cases from that same model. It allow to easily generate a large test suite from the same model or regenerate the test suite each time the system requirements change. By following this approach, the test design time is reduced and several test suites can be generated by just using different test criteria. The purpose of model-based testing is to solve problems [3] that other testing processes do not fully address like:

- Automation of the design of functional test cases to reduce the design cost;
- Produce test suites with systematic coverage of the model;
- Reduction of the maintenance costs of the test suite;
- Automatic generation of the traceability matrix from requirements to test cases.

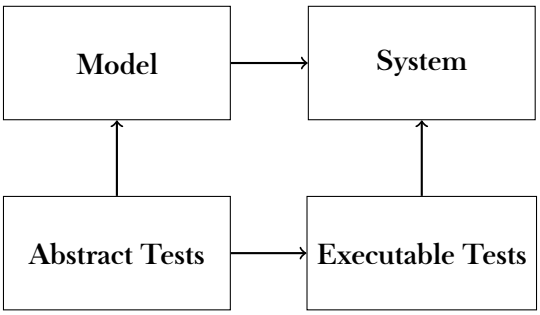

Figure 3: Model-driven definition

Summing up, the model is a partial description of the system under test, the abstract tests are derived from the model, and the executable tests are the concrete implementation of the abstract tests that can be run against the system under test.

5.1 How to model a system?

The first and most important step in modeling a system for testing is deciding the level of abstraction of the model. It should be clear which aspects of the system under test to include in the model and which aspects to omit. We should always have in mind that a smaller model is often more useful than a large and complex model, since it allow us to test each one of them independently and think about which operations should be included before writing a model for the whole system.

When modeling a system is also important to think about the data that it manages, the operations that it performs, and the subsystems that it communicates with. The difficulty of test generation is usually highly dependent on the number and range of the input parameters, so applying the same abstraction principle [6] to its input and output parameters helps to control the test generation effort.

The next step is to decide which notation to use for the model. This decision is often influenced by the model-based testing tools that are available and the notations they support. It is important to consider which style of notation is most suitable for the system under test. There are several modeling notations that have been used for modeling the functional behavior of systems [3]:

- **State-based notations**: These notations model a system as a collection of variables, which represent a snapshot of the internal state of the system, plus some operations that modify those variables. Examples of these notations include the UML Object Constraint Language, the Java Modeling Language and Spec#.
- **Transition-based notations**: These notations focus on describing the transitions between different states of the system. Typically, they are graphical notations, such as finite state machines, where the nodes represent the major states of the system and the arcs represent the actions or operations of the system. Examples of transition-based notations include UML State Machines, labeled transition systems, and I/O automata.
- **History-based notations**: These notations model a system by describing the allowable traces of its behavior over time. Various notions of time can be used, leading to many kinds of temporal logics. Although they are used as a basis for model-based testing they are better used for visualizing the tests that result from model-based testing than for defining the model that is the input to model-based testing.
- **Functional notations**: These notations describe a system as a collection of mathematical functions. Algebraic specifications tend to be more abstract and more difficult to

write than other notations, so they are not widely used for model-based testing.

- **Operational Notations**: These notations describe a system as a collection of executable processes, executing in parallel. They are particularly suited to describing distributed systems and communications protocols.
- **Stochastic Notations**: These notations describe a system by a probabilistic model of the events and input values. They tend to be used to model environments rather than systems under test. They are good for specifying distributions of events and test inputs for the system under test but are generally weak at predicting the expected outputs.
- **Data-flow notations**: These notations concentrate on the flow of data through the system under test, rather than its control flow.

State-based notations are best for data-oriented systems and transition-based notations are best for control-oriented systems [3] but the system could not be so easy to classify and it can be a mix of both data and control-oriented. Choosing the wrong notation could make the system harder to model. A good choice should the right balance between the notation tooling support and what notation suits the system best. It is important to build an accurate model, written in a formal modeling notation that has precise semantics and unambiguous meaning in order to allow different tools to understand and manipulate the model behaviour.

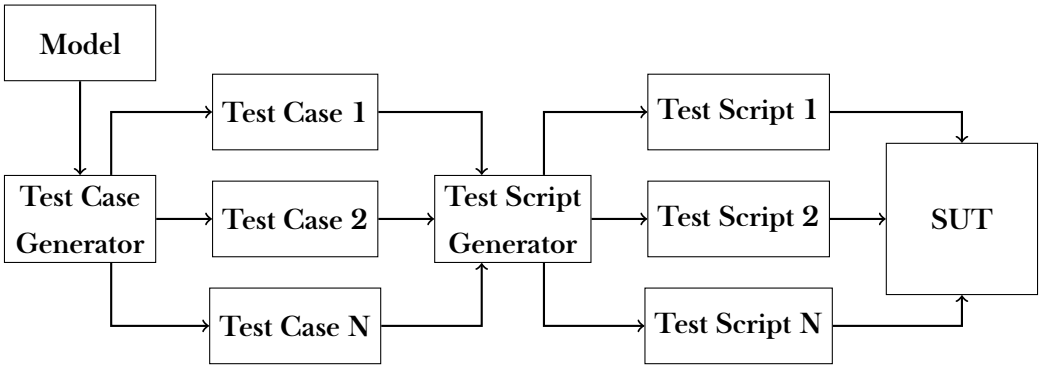

Figure 4: Model-based process approach.

After the test generation from the model and the execution of those tests against the system, each test that fails will point either to an error in the implementation of the system or to a mistake in the model. The value of model-based testing comes from the automated cross-checking between the model and the system implementation.

5.2 Online or Offline Test Generation

Online testing is where tests are executed as they are generated, so the model-based testing tool is tightly coupled to the system under test. It is particularly good for testing nondeterministic systems and for long-running test sessions, such as overnight testing based on random traversal of a model. This kind of technique is sometimes necessary if the system under test is non-deterministic, because the test generator can see which path the system has taken, and follow the same path in the model.

Offline testing decouples the generation and execution phases, so the test execution can be completely independent of the model-based test generation process since the test cases are generated strictly before they are run. It can perform a deeper analysis of the model to generate a small but powerful test suite, and the generated test suite can be inspected, use repeatedly for regression purposes, put into a test management system, and so on.

The advantages of offline testing, when applicable, are mostly pragmatic. The generated tests can be managed and executed using existing test management tools, which means that less changes to the test process are required. One can generate a set of tests once, then execute it many times on the system under test. Also, the test generation and test execution can be performed on different machines or in different environments, as well as at different times. Finally, if the test generation process is slower than test execution, then there are obvious advantages to doing the test generation phase just once.

5.3 Challenges in Modeling

Model-based testing can improve considerably the test efficiency and test quality when the models are light and available. The elaboration of a model is, probably, the key for the success of model-based testing in practice, and if done wrongly [7] can lead to a set of undesirable outcomes:

- If complex models have to be completed before testing can start, this induces an unacceptable delay for the proper test executions;
- For complex SUT, like systems of systems, test models need to abstract from a large amount of detail, because otherwise the resulting test model would become unmanageable;
- The required skills for test engineers writing test models are significantly higher than for test engineers writing sequential test procedures.

5.4 Prerequisites of Model-Based Testing

It is important to have some high-level management support for the adoption of model-based testing, as well as some enthusiasm for model-based testing among the team who will put it into practice.

Model-based testing tools implies that the test execution phase is already automated. So, for a testing team who has little experience with automated testing might be wise to gain experience with an automated approach like data or keyword-driven testing before using model-based testing.

Model-based testing models are more precise than most UML models and require detailed modeling of the system behavior. Developing these kinds of models is more similar to programming than to designing use cases and class diagrams. It is helpful to have some experience with designing interfaces and choosing good levels of abstraction.

Sometimes, model-based testing does not suit the system needs. If the system has to be tested manually because the test steps require human interaction, then the cost of that manual testing may dominate any cost savings of model-based testing.

5.5 Model-Based Testing and Agile methods

In terms of industrial adoption, model-based testing needs to be adapted to the existing testing processes that are shifting towards more agile practices from the traditional ones based on the V-model and its variations. In agile contexts, on the one hand, developers are already relying on test automation to support refactoring and generally understand its benefits as compared to manual testing.

In the eyes of the agile practitioners, writing acceptance tests that specify what the system is supposed to do, brings lot of value:

- It forces the customer to specify precisely what is required;
- When the acceptance tests pass, the customer has much more confidence on the work that has been done;
- An executable specification gives a clear and measurable objective to the development team.

Having the acceptance tests centered around a model makes not only easier to develop a significant number of acceptance tests, but also to change the model and regenerate those same tests. We can benefit even more from mixing model-based and acceptance tests if the model itself is built around some agile principles. The test models should:

- be light;
- have a precise purpose;
- grow incrementally through an iterative approach;
- encourage discussion about the exact behavior of the system;

5.6 Advantages of Model-Based Testing

Model-based testing can have a small cost-benefit over keyword-based testing and a significant cost-benefit over the other testing processes [3].

It can lead to less time and effort spent on testing if the time needed to write and maintain the model plus the time spent on directing the test generation is less than the cost of manually designing and maintaining a test suite.

The use of an automated test case generator based on algorithms and heuristics to choose the test cases from the model makes the design process systematic and repeatable and it is very easy to generate more tests.

Quantity means nothing if we loose the quality of the test suite. Several studies have shown that the use of a model-based approach tends to increase fault detection effectiveness [8,9]. It is usual to find numerous faults in the requirements and design documents as a side-effect of modeling [3]. The detection of such faults at an early stage is the major benefit of model-based testing.

5.7 Disadvantages of Model-Based Testing

In spite of all of the advantages stated above, the industrial adoption of this technology has been slow. The most common problems are the managerial difficulties, the making of easy-to-use tools, the reorganization of the work with the tools [10] and the complexity of the solutions and counter-intuitive modeling [5].

One of the fundamental limitations [3] of model-based testing is that it cannot guarantee to find all the differences between the model and the implementation, not even if there are hundreds of test cases being generated by the system model.

Model-based testing is not as trivial as other testing techniques, and requires a steep learning curve and different testing skills: modeling and programming skills. It is desirable to have a reasonably mature testing process and some experience with automated test execution before fully adopt this approach.

Up to date requirements plays a big role in model-based testing. The test cases are very coupled to the requirements, and requirements change, often. If the model is build based on outdated requirements, it will lead to unreliable results and one of these tests fails, it's hard to check if the failure is caused by the system under test, the driver, or even from the model itself. This process makes more difficult and time-consuming to find the root cause of the failing test.

Since this approach can generate huge numbers of tests, it becomes necessary to move toward other measurements of test progress instead of relying on the number-of-tests metric. Business code, requirements and model coverage are some of the alternatives.

6 Conclusion

References

[1] Mark Fewster and Dorothy Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[2] Pekka Laukkanen and Pekka Laukkanen. Data-driven and keyword-driven test automation frameworks, 2007.

[3] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[4] M. Mussa, S. Ouchani, W. A. Sammane, and A. Hamou-Lhadj. A survey of model-driven testing techniques. In *2009 Ninth International Conference on Quality Software*, pages 167–172, Aug 2009.

[5] Antti Jääskeläinen, Mika Katara, Antti Kervinen, Henri Heiskanen, Mika Maunumaa, and Tuula Pääkkönen. *Model-Based Testing Service on the Web*, pages 38–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[6] Daniel J. Mosley and Bruce Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[7] Jan Peleska. Industrial-strength model-based testing - state of the art and current challenges. *Electronic Proceedings in Theoretical Computer Science*, page 3âĂŞ28, 2013.

[8] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 285–294, New York, NY, USA, 1999. ACM.

[9] E. Farchi, A. Hartman, and S. S. Pinter. Using a model-based test generator to test for standard conformance. *IBM Syst. J.*, 41(1):89–110, 2002.

[10] Harry Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. 2013.