TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

RISTO HELINKO

GENERATING TEST CASES FROM FORMAL SPECIFICATIONS

Master's thesis

Examiner: Docent Henri Hansen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 27th September 2017

ABSTRACT

RISTO HELINKO: Generating Test Cases from Formal Specifications
Tampere University of Technology
Master's thesis, 37 pages
December 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Docent Henri Hansen
Keywords: QuickCheck, Property-Based Testing, Generative Testing, .NET Framework, Quality Assurance

Quality is a persistent problem in software development. The main method for quality assurance, testing, is a significant part of any software project. As software development processes move towards continuous integration and deployment, there is demand to increase the level of automation in testing.

Conventionally, test automation refers to automating the execution of tests. However, the test cases are written one-by-one, which can get very repetitive. This is also more costly, especially in the maintenance phase. An alternative to these example-based tests are property-based tests, where properties define *sets* of tests at once. In other words test cases are generated from these properties.

Testing targets a certain interface, such as a class, an HTTP API or a GUI. A set of properties – a formal specification – describes the desired behavior of the interface. There is always a human element to specification, because correctness largely depends on the situation and is often subjective. However, verifying that a system adheres to that specification can be automated, at least to some extent.

Test case generation is mostly used in functional programming, but this thesis explores it in the object-oriented paradigm as well. Other means of verification are also discussed, and central theoretical concepts are covered. A case study is described, where test cases were generated for the M-Files API – an object-oriented .NET API.

TIIVISTELMÄ

RISTO HELINKO: Testitapausten generointi muodollisten spesifikaatioiden perusteella
Tampereen teknillinen yliopisto
Diplomityö, 37 sivua
Joulukuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: Docent Henri Hansen
Avainsanat: Ominaisuuspohjainen testaus, Generoiva testaus, Laadunvarmistus

Ohjelmiston laatu on ongelma, joka tuskin tulee katoamaan. Laadunvarmistus tapahtuu pääosin testaamalla, ja se on merkittävä osa mitä tahansa ohjelmistoprojektia. Modernit ohjelmistokehitysprosessit perustuvat jatkuvalle integraatiolle ja käyttöönotolle, mikä vaatii testaukselta korkeampaa automaatiotasoa.

Perinteisesti testiautomaatio viittaa testien automaattiseen suoritukseen. Tällöin automaatiotasossa on kuitenkin parannettavaa, sillä testien kirjoittaminen voi olla hyvinkin mekaanista toistoa. Tämä taas tulee kalliiksi etenkin ohjelmiston ylläpitovaiheessa. Perinteisten esimerkkipohjaisten testien vaihtoehto on ominaisuuspohjainen testaus, jossa yksittäisten testien sijaan määritellään testijoukkoja. Toisin sanoen ominaisuudet toimivat testitapauksien generoinnin lähtökohtana.

Testaamisen kohteena on aina jokin rajapinta, kuten luokka, HTTP API tai graafinen käyttöliittymä. Ominaisuuksien joukko – muodollinen spesifikaatio – kuvailee rajapinnan haluttua toimintaa. Spesifikaation määrittelyssä on aina inhimillinen osuus, sillä järjestelmän virheettömyys on aina paljolti subjektiivista. Sen varmistaminen, että järjestelmä vastaa jotain tiettyä muodollista spesifikaatiota, on kuitenkin yleensä mahdollista automatisoida, ainakin jossain määrin.

Testitapausten generoinnin hyödyntäminen on yleisimmillään funktionaalisessa ohjelmoinnissa, mutta tässä työssä sitä tarkastellaan myös olio-ohjelmoinnin yhteydessä. Muitakin verifiointimenetelmiä käsitellään lyhyesti, sekä keskeiset teoreettiset käsitteet käydään läpi. Lisäksi esimerkkitapauksena tutkitaan testien generointia M-Files API:lle, joka on oliopainotteinen .NET-rajapinta.

PREFACE

My thesis process started by thinking that there must be ways to make software quality assurance more automatic. I first considered exploring static analysis and type systems, but couldn't find a way to examine these independently of programming language. This was important for me, because I naturally preferred universal techniques.

Whereas the power of types or any static analysis is almost by definition highly dependent on the language, dynamic analysis, i.e. testing, is less so. Interest in functional programming lead me to property-based testing, which could, at least in theory, be applied to source code written in any language.

Thanks to Tero Piirainen and Minna Vallius from M-Files for believing that my thesis was worth exploring and for giving me an opportunity to try generative testing in a realistic setting. Thanks also to Olli Pekkola and the rest of M-Files QA team for help in the daily work. The case study at M-Files gave this thesis much-needed direction.

Thanks to my supervisor Henri Hansen, who gave me freedom to explore my interests and was always willing to help, even when I had trouble articulating my problems.

I'm also grateful to my parents, who have unconditionally supported me through my studies. You are awesome.

Tampere, November 22nd, 2017

Risto Helinko

TABLE OF CONTENTS

1. INTRODUCTION

The objective of this thesis is to explore methods that raise the level of automation in testing. Modern software development is moving towards continuous integration and deployment, which requires frequent verification. If developers can easily verify changes, they are able to more often improve design by refactoring. More automation means that scaling – adding more tests – can be done by mostly increasing computing resources, which is easier than increasing human resources.

Verification determines whether the system is built right, whereas validation considers whether the right system is built [11]. A description of what is "right" is called a specification. In other words, verification evaluates whether a system adheres to its specification, whereas validation examines whether the specification is what the stakeholders want. A stakeholder is a person or a group who is involved with the software project, for example users, developers or management [38]. In this thesis we focus on verification, and mostly assume an informal specification exists, whether it is written down or not.

A *formal* specification is a set of properties that can be checked automatically [26]. More precisely, formality means that the specification is written in an unambiguous language including formal syntax, semantics and rules of inference. For example most programming languages meet this criteria.

Our chosen method of verification is testing, which means running and analysing the actual system. In contrast, static analysis refers to examination at compile-time, i.e. the system is *not* run, and model checking involves running a simplified version of the system. Systems usually have more possible states than can be exhaustively explored, meaning testing can give confidence but not certainty of correctness.

We also look at the system as a black box, which means we are only aware of what is visible through the interface of the system. For simplicity we concentrate on programming language interfaces such as modules, classes and functions, but many principles apply to others, as well.

There are 3 main levels of testing: unit testing, integration testing and system testing. Unit testing is about testing the smallest testable units. This is where most bugs (deviations from the specification) should be found, because they are cheapest to fix. The root cause is in the smallest possible component and testing depends on the least amount of code, so it can be done earlier in the project compared to other forms of testing.

Unfortunately, not all bugs are visible when units are tested separately. This is why integration testing is needed, in which the interaction of 2 or more components is tested. However, these kinds of bugs are increasingly hard to find, because the combined number of possible states grows exponentially as the number of separate units grows.

The final level of testing is system testing, in which the environment mimics the production environment as closely as possible. Here the possible states are even greater than in integration testing. With regards to test automation, perhaps the biggest challenge is that higher-level interfaces are less idiomatic, meaning that they conform less strictly to conventions. This makes testing harder to automate.

Improvements in quality happen if bugs are not only found, but also fixed. To facilitate this, we want to find bugs as quickly as possible after they are introduced to the system. Therefore testing should be done as often as possible during development. To make thousands of daily tests economically feasible, they have to be automated as much as possible.

Conventionally test automation refers to automatic *execution* of tests, while test cases themselves are quite mechanically written one by one. These are called example-based test, in contrast to the more general property-based tests. Instead of writing the test cases directly, property-based tests are automatically generated from the properties. The properties form a formal specification.

Test cases here can be thought of as assertions such as “foo(3) $<$ 8”. On the other hand they can be instances of properties that are more general, such as “foo(a) $<$ a + 5”. Test cases are analogous to (constant) values and properties to functions, even including the fact that functions can be values. That is, the aforementioned test case “foo(3) $<$ 8” can also be seen as a property, albeit a very specific one.

Another analogy is, that properties are sets and test cases are elements. Depending on the context, a set with a single element can be indistinguishable from a single element. All test cases are properties, but only some properties are (single) test cases.

The behavior of the system under test (SUT) depends on its input, and the results of executing the program are its output. Here "system" can mean everything from the smallest testable unit to a full system including an operating system and a set of applications. Testability of the SUT is then determined by two things: control of input and visibility of output. By controlling the input fully, we can bring the SUT to any possible state. On the other hand, the more of the output is visible, the better we can determine its correctness.

For example, while testing a graphical user interface (GUI), the best case scenario is that all behavior is accessible and visible through the UI. However, often the environment such as the operating system affects behavior as well, which makes some of the input difficult to control. On the other hand, there is functionality not visible in the UI such as sending a text message from a server.

Need for proper testability is amplified as the level of automation is raised. It is not a coincidence that property-based testing has its roots in functional programming, which emphasizes the use of pure functions. Functions here mean the general programming language construct, found in most languages. The mathematical definition of function corresponds with pure functions.

A pure function has only controllable input and visible output. Its behavior does not depend on any external state or environment, and it doesn't have side-effects, i.e. any implicit consequences not visible in the interface.

When a function depends on its environment or produces side-effects, it is not pure. These impurities are often called state, as in "a function or object is stateful", or it "has state". Having state means that the function has interaction with some global or external state at least in one direction: the function reads the state as implicit input and/or modifies it as implicit output. This global state can be a very general concept, including basically the whole universe.

To express properties about state we need modeling, which in this case refers to programming a "reference" program for the SUT to be compared to. It is the difficult part of generative testing, and often the main workload.

When general-purpose programming languages are used for modeling, the problem is never about what *can* be modeled. Modeling is programming, and the SUT is proof that the solution can be implemented. The problem is, that the model should be simpler than the SUT while still retaining its relevant functionality, because any missing functionality will not be tested. Simplicity is extremely difficult to quantify or even to define properly.

One solution would be to construct the model deliberately with different patterns than the SUT, to increase the chance that mistakes are not repeated. However, this is somewhat contradictory, assuming that the SUT has been designed with suitable methods. Then the model would be implemented in less-than-optimal ways, making it harder to be simpler.

As long as models are as expressive as any programs, this problem is unlikely to get easier. On the other hand, if something inherently cannot be expressed in the model, that has to be compensated somehow. Maybe there will be a modeling language that enforces simplicity. However, perhaps the more likely scenario is, that we just have to avoid modeling as much as possible, and use ad hoc or domain-specific techniques on the remaining parts.

In addition to finding bugs, tests also act as documentation. As such, examples can be useful, but if tests are all examples, it is up to the reader to derive any generalizations from them. Properties can more explicitly express general rules. Also, examples can be unambiguously derived from properties, but the reverse is not usually true.

Properties may be more robust than example-based tests, because they are more abstract, and therefore allow more flexibility of implementation, at least in theory. Abstraction also makes them easier to maintain, because there is less code to express similar things. Properties also scale better because increase of testing does not always require more development work.

Most modern tools for test generation utilize the paradigm popularized by QuickCheck [15], in which specification is done by defining preconditions and postconditions. This allows for a lightweight form of modeling and is also often previously familiar to developers from concepts such as Hoare triples or Design by Contract. Another characteristic of the paradigm is that the test cases are generated randomly. While briefly presenting alternatives to pre/post models and random generation, this thesis will concentrate on the QuickCheck paradigm.

The second chapter will discuss the theoretical aspect of generating test cases, divided into two main parts: formal specification and generating input. Post-test analysis is also covered briefly. In the third chapter, we review generative testing tools for the .NET Framework, which leads up to next chapter's case study. In the case study we experiment with utilizing test generation in testing an object-oriented API at M-Files.

2. GENERATING TEST CASES FROM FORMAL SPECIFICATIONS

Test case generation has two parts. The first is writing a formal specification – a set of properties. The properties can have parameters, and the set of possible values for parameters defines a set of possible test cases. These sets are usually too large to test exhaustively, so the second step is to select a representative subset. This selection of a representative subset is referred to as generating input.

Properties with no parameters are single test cases, and testable without generation of input. Therefore, even though such properties might be legitimate to test, they are trivial, and so the rest of this thesis will assume that the properties do have parameters.

2.1 Formal specification

A formal specification is a set of properties that the system under test is supposed to satisfy [26]. It is meant to be more abstract than the SUT by describing *what* the system is supposed to do, rather than *how* it is to be done [33]. A formal specification is precise in the sense that it is unambiguous and machine-readable, but it can still allow for a diverse set of solutions.

2.1.1 Testable properties

For our purposes, simply having a formal specification is not enough. We also need a way to generate and execute the test cases from it. There are helpful tools, which usually will expose an interface to help define the specification, ranging from language-specific library APIs to full-blown specification languages.

Here we will concentrate on the paradigm started by Haskell QuickCheck [15]. Tools of this type are characterized by utilization of random input generation. The simplest specifications are given as preconditions and postconditions, although more advanced options are also available [16].

The main benefit of QuickCheck-like tools is that they are usually free and open source software and besides knowing the programming language in question, require little additional expertise to start using them. There also aren't that many alternatives. For example, all tools listed in [34] have since been either abandoned or have little mentions outside marketing materials.

2.1.2 Preconditions and postconditions

Preconditions and postconditions are assertions on the input and output of the SUT, respectively. The postcondition is expected to be true, assuming the precondition is satisfied. The terminology is based on axiomatic semantics [22], later developed to a software engineering methodology called Design by Contract [27]. The idea is to form so-called Hoare triples PQR, which represents the assertion "if P is true prior to executing Q, then R will be true after its completion".

For a class in C#, for example, a specification might be a set of these triples, where each Q is a method call. There could naturally be multiple triples involving some Q.

This type of specification is also sometimes called state-based [26], which may also involve invariants, assuming there are objects or a similar concept in the language. However, the conditions do not necessarily have to mention state.

If the cost of specification is not an issue, the specification should aim for weakest preconditions and strongest postconditions possible [26]. The larger the set that fills the condition, the weaker it is. That is, the absolute weakest condition would be one that always evaluates as true.

Ideal specifications can be difficult to define, so in practice their cost has to be taken into account. This is why traditional example-based test cases define very strong preconditions when defining the input one by one. It is simply easier to come up with examples than a more comprehensive definition.

On the other hand example-based tests might have very strong postconditions because the strong precondition makes it easier to define.

2.1.3 Formalizing a specification

Here we will assume that there already is an *informal* specification of some kind. It is not always the case that there is a written description about what the system should do, but there is some process by which design decisions are made, and therefore there is an implicit specification.

The process of formalization, that is, coming up with properties, might require some learning. Often the problem is to find *easily definable* properties. These depend heavily on the situation: the language and available functions. For example, one might check a definition of a mergesort algorithm by asserting it always produces the same results as a bubblesort algorithm. This naturally assumes there is a bubblesort version that is correct, or it is trivially implemented.

Another way of specifying a sorting algorithm would be to check that the resulting elements are in order, and that all original elements are present in the result. These properties might also need some extra definitions, which may or may not be readily available and/or correct.

One novice mistake might be to look for too strong postconditions, whereas the weak ones are often easier to define. Going for weak conditions also breaks the pattern of trying to specify the function with a single property rather than with a set of them. For example, a weak post-condition for the sorting algorithm could be the assertion that the result data structure is same length as the original one. The property is true for some incorrect implementations so it is certainly not the strongest possible postcondition, but it is still useful to check.

Specification languages

Ideally the specification would not be dependent on what functions are available. We would express properties, and then separately make them computable. In practice, the amount of work needed to make them executable is a significant issue. We might even think of it in reverse – to look what can be easily expressed (with some library, for example), and use that as basis for defining the properties.

Put another way, in addition to specification we are also programming to make the specification executable. There is a trade-off, where separation of these tasks might lead to a more clear process, but seamless integration has practical advantages.

This thinking is relevant if we consider other, less language-like interfaces for specification. Unless the objective of generating arbitrary test cases can be limited, the simplicity probably does not justify the loss of expressiveness. An example of such limitation would be a need to only generate certain kind of test cases. However, it is probably not wise to develop this sort of domain-specific tools, but rather general-purpose ones.

Test-based specification

During the specification and its formalization, one usually generates and runs tests after defining any property. If tests fail, it is often caused by a bug in the property, rather than the SUT [23, 25]. This of course happens with any testing, but at least anecdotally it seems more common with properties because unlike single examples, properties can have cases that the tester did not consider.

For example, when testing a subtraction function for integers, one might define a property stating that the output is always less than or equal to the first parameter. This does not apply when the second parameter is negative.

This might seem like a downside, but it is quite the opposite. The tester plays the part of the user, and there is actually surprisingly little difference between using the SUT incorrectly and it being incorrect. This is especially true when we test internal APIs, in which case it is irrelevant for the end-user which side of the API the bug technically is.

If the API under test is public, the incorrect specification can be a sign of bad user experience. If the API is internal, it can be a potential source of bugs caused by misuse of the API.

The process of specification can also be more explicitly reversed by a tool such as QuickSpec [17]. The idea is that the tool will automatically, by using testing, generate a set of properties that will possibly hold for the implementation. “Possibly” means that they hold only as far as they have been tested. This method is still in an experimental stage, so it remains to be seen whether it will be further integrated into property-based testing workflows.

Types as specifications

Perhaps the biggest difference between describing specifications in different languages is caused by their type systems. Type-checking is a form of checking properties, and the expressiveness of the type system maps to the set of possible properties that can be defined at the type-level. In this sense test generation can be thought of as an extension to type systems. Therefore, languages with limited type systems and dynamic type-checking perhaps benefit the most from test generation.

The two main differences between these methods should be emphasized. Firstly, type systems are, at least to some extent, sound. In this context, it means that a type-checked program is guaranteed to not have type-errors at run-time [29]. This naturally simplifies things, as any property confirmed by the type-checker is more certain than its counter-part, checked by testing.

The other difference is expressiveness. Testing is not constrained by soundness or possible design flaws of the type system, meaning that there are plenty more testable properties than there are type-checkable properties. Run-time properties can be more expressive, because they have the additional information of run-time.

Expressivity has downsides, as well. With more expressivity, we are only limited by how much testing code we are willing to write. Since there are less technical constraints to stop us, it becomes more of an economical problem, which has to be solved for each case separately. In some sense it is an instance of Bjarnason's slogan "Constraints liberate, liberties constrain" [12].

Moving from type-checking to testing brings us expressivity, but it makes harder to decide what to express. It also makes it harder to determine whether or not the expressed property is actually satisfied, because there are no guarantees.

2.1.4 Properties of state

When testing stateful systems such as C# classes, we naturally wish to check properties that refer to the state. After generating a random state there is little to say about it unless we have some reference state for comparison. We call this reference state a model.

The model is a program that is somehow simpler than the SUT, and therefore easier to validate. Whereas a set of properties (the specification) is bound to be simpler than the SUT, a model as a program is much less constrained. Assuming the modeling language is Turing-complete there are no guarantees that the model would actually be any simpler.

Usually the results of a program are more important than how the results are computed. Source code that emphasizes the results is declarative, while imperative code describes what to do or *how* to compute the results. Results are described in both, but imperative code has more information.

As we mostly care about the results, we want our model to be as declarative as possible. With regards to programming paradigms, functional and logic programming are considered to be declarative, whereas object-oriented and procedural programming are imperative [10].

Unfortunately, even describing just the results usually leads to a model that is too complex. Also, the modeling languages in practice are not fully declarative, likely because compilers cannot always figure out sufficiently efficient algorithms without help from the programmer. This means that to simplify the SUT, some of the details have to be ignored.

Choosing which details are modeled and which are not, in the general case, is extremely difficult [13]. One just has to choose the ones that are considered most important, perhaps also taking into account the ease by which they can be included.

2.2 Generating input

Once there is a formal specification that can be checked for at least some subset of executions, the only thing needed is some input. There is usually some initial definition of possible input, such as all 32-bit integers or strings of arbitrary length. By generating input, we choose samples of these sets. We can optimize this in two ways: either maximizing the significance of bugs found, or the number of bugs found.

Significance is a product of severity and frequency of occurrence. Put another way, most significant bugs cause critical damage (or annoyance) and are seen often in “real” use. Severity is difficult to define accurately, and it is probably impossible to generate input on the basis of how severe bugs it would trigger. On the other hand, frequency of occurrence can be targeted by making the test data more realistic.

Testing a spell-checker would be an example of this. It takes a string of characters as input. By narrowing down the set of possible input to a set of dictionary words, we are making the input more realistic. Therefore any bug found is probably more significant than with completely random strings. The problem is that now some input is excluded, and likely some bugs, too.

The other option – optimizing for the largest number of bugs found – is also difficult, especially in the general case. Predicting people’s mistakes might be impossible, because any patterns being recognized can cause corrections and perhaps over-corrections. We also design new tools to mitigate and prevent these mistakes, while possibly creating new avenues for errors.

In addition to defining the input set, assuming we cannot test it exhaustively, we also need a generating algorithm, which in this case is perhaps more clearly a selection algorithm.

If the specification is given as some sort of structure like a Finite State Machine, the concrete input can be defined by structural coverage criteria that narrows down the inputs [34]. Examples of such criteria would be “all states” or “all transitions”. The actual inputs these lead to depend on the situation and the specific algorithm. In this paradigm a model acts as both an oracle (helping to express stateful properties) and as a basis for input data. Tools of this kind have yet to break into the mainstream, and therefore we will focus on the other methods.

A problem with these structure-based approaches is that a structure is needed, which is extra work for the tester. Also, to understand these tools deeply, one would need to understand the selection algorithms.

The most common selection algorithms make use of randomization. In practice it means a pseudo-random algorithm, which is unknown to the user. Usually its distribution is known, that is, by which probability each element is chosen. The main reason to use randomization is to avoid making assumptions with the selection algorithm, although there are *some* assumptions such as the distribution.

An interesting alternative to the QuickCheck way of random generation is SmallCheck and its sibling Lazy SmallCheck [31]. Instead of a sample of n values, they use the n smallest values. Of course, for some (user-defined) types this requires a definition but such is the case with random generation, as well.

2.2.1 Generating random values

Results of randomized algorithms depend on not only their input, but also on values from a random-number generator (RNG) [18]. RNGs are strictly speaking usually pseudorandom, because computers are deterministic and unable to produce true randomization. However, for our purposes this pseudorandomness will suffice.

If a fully deterministic (non-random) algorithm were used for selection, the generated test data would always be the same. If that data was based on some mistaken assumption, the mistake would be repeated at every execution. By having different data at every execution we can avoid at least some of these assumptions.

Repeating the same tests regularly might also have some advantages, but perhaps less than it would first seem. For example, if a test fails for some build but passes for the previous one, we can assume that changes between the builds contain the cause. If test inputs vary, we don't have this kind of history to examine. However, the previous build is usually still available, and having that failing test input, we can run the exact same test for the previous build as well.

Choosing data optimally might be impossible especially over time, because as systems develop and change, so do the inputs that trigger bugs in them. With limited resources, decision to test with some data always means that some other data is left untested. It would be best to make as few of these decisions as possible, and in a way, randomization allows that. This is also an advantage in terms of user experience of the testing tool. As there are less choices, there is less work for the user.

The only choice is the distribution, and even that has a reasonable "default" option: a uniform distribution. On the other hand, the distribution can be chosen in a more advanced way. If the realistic data can be modeled statistically, we can find compromises between the two aforementioned optimization tactics.

For example, the spell-checker input could consist of the following: 60% dictionary words, 30% alphabetic strings, 10% fully random strings of all Unicode characters. Domain knowledge and statistical expertise will help with defining effective distributions.

2.2.2 Generating random objects

In addition to input values, the behavior of a program can also depend on an external state. We will use stateful objects as an example, but the methods are, at least to some extent, applicable for state in general.

By stateful objects we mean Abstract Datatypes [21] (not to be confused with Algebraic Datatypes). As users of the interface (the SUT), we are not aware of the actual types of these objects, which means we cannot generate them directly. The state of the objects is accessed via special functions, usually called methods. We can then generate the objects with a sequence of calls to these functions. Minimally this would be just a call of the constructor.

The problem is, that we cannot assume that an object constructed with some value a would be equivalent to an object constructed with some other value, and then modified to contain the value a. By definition the behavior of a stateful object is determined by its history [9]. The objects shouldn't be generated with just the simplest sequence of calls, but with a randomly generated sequence. Similarly to generating values, the sequence of function calls can also be made to statistically represent some approximate user profile.

2.2.3 Preconditions and generators

In the QuickCheck paradigm, there are two modes for defining input: preconditions and generators. For our purposes a generator can be thought of as a function that produces values, even though precise definitions depend on the language: in QuickCheck they are typeclasses and in FsCheck classes. Preconditions and generators differ in two ways, semantically and execution-wise.

The semantic difference is that a precondition is specific to a certain property, whereas a generator defines how to produce input of some type. The generator is run first, after which the value may or may not be discarded based on the precondition. This can lead to issues with efficiency, if the generation process is costly and the precondition is strict enough. The precondition can be always integrated into the generator, but it's bothersome (both for the implementor and the reader) to define a generator for each property, so a balance should be sought.

Another issue with execution can arise with languages that have eager evaluation, meaning most languages except Haskell. In FsCheck, the library evaluates each postcondition before checking the precondition [32]. Therefore, if the property is nonsensical without the precondition, such as a division by zero, the property fails due to an exception being thrown. However, lazy evaluation can be forced for a postcondition in FsCheck.

2.3 Post-test analysis

After testing, there usually needs to be two kinds of analysis: assessment of coverage and investigation of failures. Coverage is a measure of how much of the code was executed during testing. Because the specification and models are more abstract and indirect than conventional tests, it is perhaps even more important to measure coverage. However, as this analysis happens completely at runtime, the measuring process is likely the same for both generated and hand-written tests. Therefore we consider it to be outside the scope of this thesis.

On the other hand, investigating failures can be quite different when the tests are generated rather than hand-written. Random data has the problem that it produces random counter-examples. In QuickCheck, the problem is mitigated by a process called shrinking [14].

Shrinking is a basic procedure that after a test has failed, will attempt the same test with ever smaller input until if no longer fails. The size of user-defined types is defined as a shrinking function, where applicable.

For example, in QuickCheck, given a value, the shrinking function returns a list of elements of the same type [14]. These elements are in ascending order with regards to size. QuickCheck then tries each one, and greedily chooses the first one that makes the property fail. It then attempts to shrink again with that new value, until there are no more fails.

There are no guarantees that the shrunk example is the smallest. It also depends on the type and its shrinking function. For example, if a property fails with integer values 4 and 20, and the original failing case is 20, the default shrinking function of QuickCheck will not be able to successfully shrink. This is because 'shrink 20' returns the list {0, 10, 15, 18, 19}, which for the example contains no failing values. The guarantee would obviously be possible, but has apparently not been considered worth the cost by the QuickCheck developers.

Another, more advanced approach is to attempt to generalize the counter-example. This is algorithmically quite a bit more demanding task and its effectiveness is not yet been established. However, there is a working implementation for Haskell in SmartCheck [30].

After shrinking, the failing test case is (hopefully) similar to an example-based test case. Further debugging is not affected by the tests being generated, and therefore actual bug-fixing is considered to be outside the scope of this thesis.

3. REVIEW OF TOOLS FOR THE .NET FRAMEWORK

In this chapter we review tools for the .NET Framework, specifically with the intention of testing M-Files software. In this case the specifics of M-Files did not effect significantly the choice of the testing tool, other than some need for support on the .NET Framework.

3.1 Choosing a tool

Criteria for tool selection can be divided in two parts. First of all, no tool can generate tests for an arbitrary interface. Therefore there are limitations to which back-ends a given tool supports out-of-the-box. These back-ends are usually programming languages or graphical user interface (GUI) testing frameworks such as Selenium.

If the tool does not support the desired back-end, it might be possible to implement an adapter to add the support. This demands non-trivial level of commitment to a tool without a proper trial, which is a risk. In practice, many tools can be ruled out on the basis of back-end support.

There also might be peripheral tooling that needs support, such as a continuous integration service. These integrations are usually less important and easier to implement than the adapters related to the actual generation.

The previous factors are related to how the tool fits the SUT at the time of evaluation. This should also be assessed in the future tense. Two main aspects can be recognized: the license and the activity of the project.

If the tool is a closed source, proprietary product, the company is of the most interest. Can they go bankrupt? Can they stop development and drop support for the tool? Accurate answers might be difficult to find, but the size of the user base is likely a clue, and can be estimated.

For open source software (OSS) with a permissive license, the questions relate more to the number of users and developers. Especially deducing the number of developers is more direct for OSS, because one can see the exact number of committers from the repository. Also the number of people asking questions can give estimates about non-developer users.

The second set of criteria concerns the UI of the testing tool itself. The specification is given in some formal language, but those languages can look quite different. They might have graphical representations such as Statecharts, be purely textual like conventional programming languages, or both.

Main avenues of assessment for the specification languages are their suitability for specification (in general or regarding a certain domain) and prior skills of the testers. The problem domain and therefore characteristics of the SUT can inform the choice of language, but the relationships are not straight-forward. One coarse guideline could be, that a data-oriented system is easiest specified by pre/post notation, whereas a more control-oriented one would benefit from a transition-based approach [34].

The correspondence between modeling paradigms, specification languages and tools are not one-to-one. A tool might support multiple languages [6, 16], and a language might also support multiple modeling paradigms.

A familiar specification language will be easier to start using. Common programming languages, such as C#, are more likely to be previously familiar. The trade-off is that such languages will probably be less than ideal for modeling and specification.

Testers with little programming background might prefer languages with graphical representations, or even programs that have a “non-language” interface. Any interface can be seen as a language, though, even something that only has a single button. The language in that case is just very constrained and non-expressive.

However, due to reasons discussed in the previous chapter, it is usually most convenient if the specification is done with a programming language.

3.2 FsCheck

FsCheck is an open source generative testing library for .NET [4]. It is a re-implementation of Haskell's QuickCheck [15,16]. FsCheck is written in and primarily developed for F#, but specifications can also be written in C# and Visual Basic.

3.2.1 Support and license

FsCheck supports any SUT that exposes a .NET API. It has a BSD 3-clause license [5], which is one of the most permissive open source licenses. The license allows commercial use, modification and distribution, on the condition of the license and copyright notice.

FsCheck is ran as an independent open source project, so it doesn't have a paid support system. It is relatively simple, though, and one can get started with the documentation, ask questions via the Github issue tracker and as a last resort, take a look at the source code. There are not many FsCheck-specific tutorials out there, but if the scope is broadened to any *Check library, there should be enough.

There is decent support for CI services, namely NUnit and xUnit.NET [4]. There is also a plug-in from the behavior-driven testing framework SpecFlow to FsCheck [28].

Development of FsCheck is on-going and it seems to have enough developers and users to be a viable long-term choice. The Github repository shows 50 contributors and 457 “stars”. For comparison the popular unit testing framework NUnit has 108 contributors and 1102 stars at the time of writing [7].

3.2.2 Specification languages

As mentioned previously, specification and modeling in FsCheck can be done with F#, C# and VB.NET. F# is a functional language in the ML family of languages. Being functional it is also more declarative than object-oriented languages such as C# and VB.NET, and therefore the closest one to formal specification languages [10]. For people with only imperative programming experience from popular languages such as C, C++, Java, JavaScript and Python, C# will probably be the most familiar.

The multi-language support is a part of the .NET framework rather than FsCheck itself, so it is somewhat more robust than an ad hoc solution would be. Due to this interoperability, it also doesn't matter which language the SUT API is in, as it should be similarly accessible from any of the specification languages.

The documentation and user base lean towards F#, and since FsCheck is written in it, any extensions made are likely easier to implement with F#.

In conclusion, previous familiarity might favor C#, where as F# has better support in the user base and is more suitable for specification.

3.3 SpecExplorer

SpecExplorer (SE) is a tool developed by Microsoft, first released in 2004 [35]. It is designed for model-based testing of reactive object-oriented software. The models are written in Spec#, which is an extension of C#.

However, the last release of SpecExplorer dates back to 2013, when support for Visual Studio (VS) 2012 was added [8]. VS has since had major releases 2013, 2015 and 2017. There is no indication that Microsoft would add support for these versions.

SpecExplorer is fully integrated with VS and is not usable without it. It is also fully proprietary, which means no other organization can pick up maintaining it, unless Microsoft decides to let them. The tool itself might be promising, but due to these restrictions it is currently unusable.

3.4 NModel

NModel is a model-based testing tool that seems to originate from Microsoft Research, even though its origins do not seem to be explicitly stated anywhere. However, it is released under a "Microsoft license" [2], which seems to be a somewhat standard open source license with possible exceptions related to patent claims.

Compared to SpecExplorer, NModel is more loosely coupled with other tools, which means it doesn't need a new release for every Visual Studio release. In that sense, it would certainly be possible to use it. However, the problem is the lack of user base.

Aside from some research activity [19, 20, 36, 37] and a book [24] around 2008, there seems to be no signs of anyone actually using it. The only release at Codeplex is also dated 2008 [1], and there are basically no signs of further development anywhere else, either.

Inactivity of the project is an indication of the quality of the tool. Even if the tool has been misjudged by the community, an active user base is required for support and maintenance, and therefore NModel is not a viable choice.

4. EXPLORING TEST GENERATION AT M-FILES

4.1 Background

We set out to investigate the viability of test generation at M-Files. First we explored the available tools, and then applied one of them in a proof of concept, to determine whether it was a cost-effective method to improve quality in this case.

4.1.1 Overview of M-Files

M-Files is both the name of the company and its main product. The product is a system for Enterprise Content Management (ECM). Content here means all kinds of documents and files that businesses may want to save and distribute (mostly internally), such as customer information or instructional material. There are also features related to reporting, access management, collaboration, search, and version control.

Content is saved an M-Files Vault. Companies usually have several vaults to ease restricting access and to keep the content relevant. Pieces of content, such as documents, assignments and videos, are called objects. To separate them from C# objects, we will call them MFObjects. They may or may not have files connected to MFObjects. For example, an assignment can be associated to a relevant file such as a Word document, or it can be completely 'stand-alone'.

MFObjects have properties. There are built-in properties such as class and type, and users or administrators can add them, as well. Properties have values: an MFObject's class might have the value 'document'. The properties represent most of the metadata that many other features are built upon.

4.1.2 General architecture and technologies

The main components of M-Files are the client, server and administrative applications. There are separate clients for Microsoft Windows (desktop), Android, iOS and web browsers. The M-Files server can be run either in company premises or in an Azure cloud service. A graphical desktop application called MFAdmin can be used to configure the server.

The main influence behind our choice of testing target was that we wanted to expend most of our resources to exploring the testing technologies. In other words we aimed to minimize time spent learning the system under test. This meant good documentation and relative simplicity were first priorities.

Most of the M-Files codebase runs on Microsoft .NET Framework. Other possibilities of test targets included HTTP APIs and GUIs, but due to available tools and our limited resources we soon decided to concentrate on the .NET interfaces. Of those, the best documentation was about the public API of the M-Files server called simply the M-Files API.

M-Files API

An important part of M-Files is client-side extensibility. Additional customer requirements – extensions and modifications – can be implemented by in-house personnel, but also third-party consultants. For this purpose, much of the M-Files server functionality is exposed through the M-Files API [3].

When the development of M-Files was first started in 2002, the .NET Framework had just been released. Many of the problems now solved by .NET were previously handled by the Component Object Model (COM), which ended up being the basis of M-Files as well. The two are somewhat compatible, but limitations of COM can still be seen in the M-Files API. For example method overloading and constructor parameters are not supported.

4.1.3 Issues with current QA process

Quality assurance at M-Files is mainly done by the two QA teams of about 20 engineers in total. There are two sorts of automated tests: unit tests of the M-Files

API and UI tests of clients (desktop, web and mobile). Most of the resources are spent on different kinds of manual testing. There is also extensive internal use of development versions.

As the industry in general, M-Files is moving towards Continuous Delivery (CD), which means that certain tasks in the development process, including testing, need to be done more often than before. Therefore, it would be valuable to automate the tasks as much as possible.

M-Files has also reached a level of maturity that has started to cause new problems. The upcoming M-Files 2018 release includes a feature set called Intelligent Metadata Layer (IML), which has demanded more changes in the existing codebase than previous features. This means that in addition to testing the new features, there is more need for re-testing the older features than with previous releases.

4.1.4 Selecting the tool

We started by a general survey of the tools to see what is available. Management hoped that the tool would allow testers to model the software without extensive programming skills.

Initial resources for the case study was one person for 4-6 months, after which we would hope to have a good sense of feasibility. This meant that most of the time was needed to learn the tool and experiment using it with M-Files source code. In other words, the tool needed to work “out of the box” on the .NET Framework. This narrowed down the options to three: SpecExplorer, NModel and FsCheck.

SpecExplorer was ruled out after finding out that it was not supported in latest versions of Visual Studio, which is an integral part of developing M-Files software. NModel was similarly not being developed anymore, and its user base was non-existent.

FsCheck, on the other hand, was fairly actively developed. Support-wise it was mostly neutral, being an open source project. There are no paid support options, but on the other hand there are no barriers to learn and develop the tool independently.

Since there were so few options for tools, we ended up not defining specific requirements. FsCheck was the only viable choice, so from this point on the question became, whether it could be used to improve the M-Files QA process.

4.2 Generating tests for the M-Files API

FsCheck has its roots in the world of functional programming, which means it concentrates on testing pure functions. There were basically no pure functions in the M-Files API, which meant we had to rely on the more advanced features of FsCheck.

After gaining a basic understanding of FsCheck, we set out to model the most basic operations of M-Files: create, read, update and delete (CRUD) of MFObjects in a Vault. Update in this case means to modify the properties of an MFObject.

At the time FsCheck had two interfaces for modeling – Command and Machine. The latter was still in an experimental stage but had more features. It had also already been decided that Command was going to be deprecated, so we decided to use Machine.

We chose to use C# as our modeling language due to previous familiarity within the company. It also seemed that using the same language in modeling and elsewhere in the codebase could be an advantage.

4.2.1 Modeling the state

The simplest model we used was a single integer denoting the number of MFObjects in the vault. As we started to model more features, it was useful to define them as classes instead of built-in data types. Then we could more flexibly add different kinds of fields and methods for them.

Modeling an object-oriented API with objects obviously feels natural but at some point starts to raise important questions. When the languages for modeling and implementation are the same, the risk of replicating the implementation and its bugs is increased. The languages are also at the same level of abstraction.

Put another way, simplification cannot be done on the language-level. There are then two remaining approaches. Firstly the model can be less efficient computationally. This is mostly useful if the implementation of the SUT is complicated by optimization. The other approach is to leave out implementation details. The choice of what to model is difficult because non-modeled features are not tested. However, that is the situation with all testing, because not everything can be tested.

4.2.2 Setting up the SUT

At first we experimented with having two options for initialization – an empty vault and a 'sample vault'. The sample vault is delivered with the trial version of M-Files. It is significantly more complex compared to an empty vault: it contains a few hundred MFObjects and about 3 times the property definitions than an empty vault.

This metadata structure complicated the definitions of operations later. There were two choices to implement this: either to hard-code the structure or extract it from the SUT programmatically. A programmer's instinct would be to avoid hard-coding, but reading the state from the SUT into the model would blur the lines between the SUT and the model.

The integrity of the model relies on the fact that it is separate from the SUT. The extracted state may contain assumptions that were not intended. For example, if there are 400 MFObjects initially, then empty vaults are never tested. Perhaps the biggest problem is, that these assumptions are then not visible in the model code, but in the preconstructed vaults somewhere else.

In addition to the variety of initial MFObjects, also the number of them was a double-edged sword. Every search was now iterating through 400 MFObjects more than 'normal', but they were the same MFObjects every time. Test runs took longer, but it wasn't clear whether we were getting a return for that time.

A sample vault seemed to give us complexity for free, but it was always the same complexity. We also couldn't leverage all that complexity without manually coding it into our setup of the model.

A blank slate meant that as much of the state as possible was in our control. It made the code more like a specification and less like a test script. Therefore we decided to go with the minimal empty vault.

4.2.3 Modeling the operations

The operations are also modeled as classes, but unlike with the state, the interface is defined by FsCheck. We first started with modeling the operation of creating the MFObjects.

When creating an object through the M-Files API, one has to decide which class it belongs to. We first tried randomizing this, but creation of the MFObjects depended on the class and made randomization complex. Specifically, one of the four built-in classes for MFObjects had a required property, which meant it had to be dealt with separately.

This started to look like another version of the problem with the sample vault. We were using the existing metadata structure, and not defining our own. Therefore we chose to ignore the special classes, at least for the time being.

Uniqueness of operations and MFObjects

As mentioned before, random generation is actually just random *selection* from a finite set. In this case, the generator is defined by a set of classes that implement Operation interface in FsCheck [6]. However, depending on the chosen generator type, FsCheck may or may not instantiate the operation each time it is selected.

Specifically, if the generator is defined with the method call Gen.Fresh, each operation can be a separate instance of the class, and can therefore be unique. Whereas other generators are defined with a set of Operation objects, Gen.Fresh is defined with a set of functions that return an Operation object. By instantiating the Operations in these functions, the set of selection becomes, in some sense, infinite.

We had to make a decision about this quite early on, without much knowledge of how it would affect subsequent modeling. If there were some objectives about which properties of the MFObjects we wanted to check, they could have informed this decision. However, we were mostly after low-hanging fruit and simply checking properties that came easy.

Using unique MFObjects seemed to give us most freedom later, so we went with Gen.Fresh. The idea was that after creating and modifying random MFObjects, a read operation could pick a random one (or go through all) and compare the model version to the one in the SUT. However, being able to pair the corresponding MFObjects turned out to be harder than originally thought.

Identifying MFObjects in the Vault

When an MFObject is created through the M-Files API, the method returns an integer ID, by which the MFObject can later be accessed. MFObjects can naturally be searched by name or other metadata as well, but the ID is the only thing guaranteed to be unique.

It was somewhat surprising that we couldn't just use the ID without assuming that we know before-hand what it will be. This is similar to the problem we had during the setup phase earlier. The correct way is to construct the model and run the SUT correspondingly, but not the other way around. That is, data should flow only from the model to the SUT, not from the SUT to the model, and this applies to things like IDs as well. It could be done, but it's clearly not intended and would lead to a messy model at best.

We then chose to use names as our identifiers instead. Uniqueness was trivial to achieve by using a built-in method from .NET that generates unique IDs.

Even though it first seemed otherwise, the flow of data even in the "correct" direction is still somewhat messy to implement. The only way we came up with was to have internal state for the operations. Naturally here the separate instances, i.e. using Gen.Fresh, is mandatory.

Using internal state assumes that the method Operation.Run is run before Operation.Check, which is not ideal but at least it is documented behavior [6] and therefore less likely to change.

Generating data during setup vs. at runtime

Having run into similar problems twice raises the question of its cause. Why did it feel necessary to have data flow between the model and the SUT?

First of all, there was the choice of when to generate data. It seemed definitely easier to do at runtime, because then data can be generated purely by need. Especially since the model itself is unaware of the size of test sequences and the number of them, this seems like the more elegant option.

However, when we chose to generate data at runtime rather than during setup, we had an implicit assumption that the data points should be unique. This would make the data more random, but that does not always make it better. In fact, real data based on natural language definitely has more collisions than random strings of characters, which might prevent detecting bugs.

If this assumption is dropped, the extra information gained by delaying the generation to runtime is not that beneficial. On the other hand, the separation of the model and the SUT is much more natural when as little as possible is done at runtime.

Modeling the reading operations

Defining the modifying operations such that MFObjects were unique and identifiable made checking properties almost trivial. We had checks on two levels. On the Vault-level we simply counted MFObjects. On the object-level we picked a random MFObject from our model and checked that it had a corresponding MFObject in the Vault.

4.2.4 Running the tests

By the time we reached the point of trying out longer runs, the amount of work needed for modeling had already become apparent. This meant we did not want to spend time developing the model anymore. If we were able to generate and run some of the tests regularly with next to no effort, we would do it, but otherwise we would have to abandon the idea.

The first problem was to scale the runs appropriately in terms of execution time. This was done simply by timing a few test runs. There were some operations that were dependent on the size of the vault. For example counting the MFObjects had to be done by a blank search, but even that was most likely just a linear time operation.

For reference, a set of five runs with a maximum size of 500 took about 13 minutes, whereas with 2000 it took 210 minutes. The full number of operations in the first was about 1500 and 6000 in the second. This means the operations clearly weren't constant time, but more accurate analysis would have needed more runs.

The first overnight run lead to a failure due to an undocumented restriction on the API method used for all searches. It always returned a maximum of 500 results, which obviously demanded longer runs to surface. The real problem was actually that this method should have been marked as deprecated, as there was another search method that had a parameter for maximum number of results.

This was a relatively trivial to fix, but it did show that the tests weren't as robust as they needed to be. The longer runs also demanded more from logging because reproduction of failures was so slow. Especially shrinking a sequence of, say, a 1000 operations takes a long time because the first level itself needs a 1000 runs, including setting up the SUT each time.

4.3 Experiences

4.3.1 Testing the M-Files API

Structure of the API

When the level of automation is raised and more formal properties expressed of the API, the need for regular structure is even more important than otherwise. The M-Files API, however, caters mostly to extension scripts so re-structuring has not been a priority. Also the process of development has been mostly adding methods whenever one is needed, rather than systematically designing and periodically refactoring.

For example, to "get" something sometimes means an MFObject, sometimes an ID. The ID might be of type long or type int, further complicated by the fact that these types have different meanings in languages even inside .NET. The API also doesn't follow a particular language in terms of conventions, so sometimes "get" is a method and sometimes it is implemented as a property.

These discrepancies mean that the model also has to be written in a relatively ad hoc manner. Of course it is not merely a modeling issue, but also a more general testability and usability problem.

Documentation and lack of deprecation

Even though our test runs were fairly limited, there were two legitimate failures during testing. Both were in some sense misuse of the API, but they could have been easily prevented by better documentation and compiler warnings.

The first failure concerning search was described in the earlier section. The other failure was a similar issue, in that it also was already fixed by a new and improved method, but the old method was still available with no indication that it should not be used.

These old methods should clearly be deprecated properly, by marking them in documentation and having them produce compiler errors when present in code.

Constructing objects

In the M-Files API there are no parameterized constructors. This means that initializing the object is removed from the creation of it. There also isn't any other idiom to initialize things in a consistent way. So whenever you need an object, you have to figure out how to get it. This often leads to chains, such as object A needs an object B, which needs an object C etc. When each object is obtained differently, this is actually a large part of the coding effort.

4.3.2 Using FsCheck

FsCheck as a tool seemed flexible, lightweight and relatively easy to understand. Since the whole idea of generative testing is more suited to functional programming, using C# got in our way somewhat. Therefore perhaps using F# would have lead to a more fair assessment of the tool itself.

C# as a modeling language

After getting used to functional languages such as Haskell, C# was somewhat frustrating to use. Models should be as high-level as possible, concerned with the denotational rather than the operational aspect of the program. With C#, however, there was often unnecessarily verbose definitions, which becomes especially annoying when experimenting and therefore constantly making changes.

All the state models ended up being product types, meaning they were a compound of other types. For example a Vault consisted of a list of type Models.Object and a list of type Models.PropertyDef. Implementing these as classes is annoying because all the constructors are completely trivial, listing all the parameters and assigning them to the properties, but they still have to be written every time.

The syntax for generics in C# is also needlessly verbose, leading often to confusing nested declarations with the angle bracket notation. Many higher-order functions such as map and fold are included in the LINQ package which has its own peculiarities.

C# has immutable lists as a library implementation, but the syntax is quite complex. In Haskell, for example, all lists are immutable,and they are denoted by enclosing elements in square brackets. An empty (immutable) list in this notation is simply a pair of square brackets. In C# the corresponding value is written as “ImmutableList<SomeType>.Empty”.

Even though modern C# has these functional features, it is not designed for them. It is fundamentally still an imperative language. The strengths of imperative languages – possibility of optimization and concreteness – are mostly irrelevant for modeling. On the contrary, to facilitate validation, we would prefer to express the models as abstractly as possible, while computational efficiency is not a priority.

5. CONCLUSIONS

Our main conclusion was that generating test cases was not suitable in the current QA process at M-Files. The M-Files API was simply deemed too large, irregular and object-oriented (as opposed to functional).

5.1 State

State is a significant issue when generating tests. For impure functions state is implicit input and output which cannot be controlled in a consistent way – at least not as consistent as mere arguments and return values. When something of interest is happening concerning the state, we have to accommodate it in the state model. In an object-oriented API such as the M-Files API, *most* of what happens, concerns the state. This means that when a new property is defined, more often than not, the model has to be extended.

It might help if the API was extremely uniform in how different classes are constructed. Then one might be able to reuse more of the model code. This would likely also improve usability, but it is hard to assess whether it is worth massive refactoring.

Not all functions have to be pure for generative testing to work. It is mostly problematic when using state is the default way of implementation, and not the other way around. Using external state should be considered a trade-off that often makes implementation easier, but is more difficult to verify than its pure counter-parts.

The difficulty stems from the fact that there is strictly more that an impure function can do compared to a pure one. Then again, basically anything can be implemented by pure functions, as any non-trivial Haskell program will show, because it has no impure functions.

By decreasing the number of stateful functions in the SUT, the state model will shrink, and with it, the bulk of the work related to test generation.

5.2 Size of the API

The M-Files API has hundreds of methods. A well-designed API would have the minimum number of artifacts (classes, objects, methods) to achieve the maximum amount of functionality. The problem is, that the M-Files API isn't really designed. It is simply extended by need.

Methods are added in two ways: to accommodate new functionality, and to improve existing methods. Breaking changes are avoided, so improvements are also additions. While the users of the API can ignore redundant methods, the testers cannot, unless the old methods are deprecated and eventually removed.

While avoidance of changing the API is desirable, *never* changing the API is an exaggeration that will eventually be detrimental. The improvements never permeate fully to the users, and resources to enhance quality are diluted because of faster growth than otherwise.

It is also questionable user experience to have redundant methods. At the very least, when a new method for something is added, it should be noted in the documentation for the previous one.

BIBLIOGRAPHY

[1] Codeplex: NModel. https://nmodel.codeplex.com/.

[2] Codeplex: NModel license. https://nmodel.codeplex.com/license.

[3] Documentation for the M-Files API. https://www.m-files.com/fi/api.

[4] The FsCheck Github repository. https://github.com/fscheck/FsCheck.

[5] The FsCheck Github repository: license. https://github.com/fscheck/FsCheck/blob/master/License.txt.

[6] FsCheck online documentation. https://fscheck.github.io/FsCheck/.

[7] The NUnit Github repository. https://github.com/nunit/nunit.

[8] Visual Studio Marketplace: SpecExplorer. https://marketplace.visualstudio.com/items?itemName=SpecExplorerTeam.SpecExplorer2010VisualStudioPowerTool-5089.

[9] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[10] Sergio Antoy and Michael Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, April 2010.

[11] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[12] Runar Bjarnason. Scala World 2015: constraints liberate, liberties constrain. https://youtu.be/GqmsQeSzMdw.

[13] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[14] Koen Claessen. Shrinking and showing functions: (functional pearl). *SIGPLAN Not.*, 47(12):73–80, September 2012.

[15] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000.

[16] Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Not.*, 37(12):47–59, December 2002.

[17] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: Guessing formal specifications using testing. In *Proceedings of the 4th International Conference on Tests and Proofs*, TAP'10, pages 6–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[19] J. Ernits, M. Kaaramees, K. Raiend, and A. Kull. Requirements-driven model-based testing of the ip multimedia subsystem. In *2008 11th International Biennial Baltic Electronics Conference*, pages 203–206, Oct 2008.

[20] Juhan Ernits, Margus Veanes, and Johannes Helander. Model-based testing of robots with nmodel. In *TestCom/FATES 2008 Short Papers*, June 2008.

[21] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages, 3rd Edition.* The MIT Press, 3 edition, 2008.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[23] John Hughes. Testing the hard stuff and staying sane. `https://youtu.be/zi0rHwfiX1Q`, 2014.

[24] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#.* 1 edition.

[25] Jessica Kerr. Property-based testing for better code. `https://youtu.be/shngiiBfD80`, 2014.

[26] Axel van Lamsweerde. Formal specification: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 147–159, New York, NY, USA, 2000. ACM.

[27] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[28] Gaspar Nagy. The specflow.fscheck github repository. `https://github.com/gasparnagy/SpecFlow.FsCheck`.

[29] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[30] Lee Pike. Smartcheck: Automatic and efficient counterexample reduction and generalization. *SIGPLAN Not.*, 49(12):53–64, September 2014.

[31] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. *SIGPLAN Not.*, 44(2):37–48, September 2008.

[32] Kurt Schelfthout. FsCheck online documentation: Properties. `https://fscheck.github.io/FsCheck/Properties.html`.

[33] J. M. Spivey. *The Z Notation: A Reference Manual.* Oriel College, Oxford, OX1 4EW, England, UK, 1998.

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

[35] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. *Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer*, volume 4949, pages 39–76. Springer Verlag, January 2008.

[36] Margus Veanes, Juhan Ernits, and Colin Campbell. State isomorphism in model programs with abstract data structures. volume 4574, pages 112–127. Springer Verlag, June 2007.

[37] Margus Veanes and Wolfram Schulte. Protocol modeling with model program composition. volume 5048, pages 324–339. Springer Verlag, June 2008.

[38] Karl Eugene Wiegers. *Software Requirements.* Microsoft Press, Redmond, WA, USA, 3 edition, 2013.