A Formal Analysis of Iterative TDD

Authors

Hemil Ruparel (hemilruparel2002@gmail.com)

Nabarun Mondal (nabarun.mondal@gmail.com)

To be submitted to *IEEE Transactions on Software Engineering* (https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32)

About

This document was created over authors discussing measurable efficacy of modern development practices. These discussions triggered a now famous LinkedIn post which got eventually shared by Kent Beck. Arguably, the formalism in that post needed a lot more work. In this paper we try to semi formally present the argument for Test Driven Development (TDD) and the reason behind it's efficacy. We also caution against the cargo cult development which is happening in the name of TDD which can be demonstrated to be not effective.

Abstract

We first introduce some concepts. Following concepts gets formally defined :

1. Functions as Point Mapping
2. Software as functions
3. Tests as higher order functions
4. Equivalence Partitions
5. Coupling in Software

Based on these we argue about the nature of the software development in terms of TDD and TDD gets formally defined for the first time in the history of TDD.

In the next section we showcase the context in which TDD provably works.

We then argue from current development trends that TDD is definitely not being used in such narrow provable context.

We conclude by stating while evidently it can not work except in narrow contexts, that is the best tool we have when applied in those right contexts - e.g Unit Like Tests where implementation has almost non existent coupling.

Definitions

Specification of Functions via Point Pairs

Any function, computable or not, can be imagined to be pairs (potentially $\aleph_1$) of input and output points in some abstract space. It makes sense to describe functions by defining their specific outputs at specific points or a large set of equivalent points. This list of pairs we shall call point specification or "specification" for brevity.

For functions which are well behaved this makes some sense. But even for well behaved functions this is not a good enough approximation.

Take a nice function like $f(x) = x$, a tautology but one can not define this function by adding pairs of specification values. A much more interesting function like $f(x) = sin(x)$ is much harder to describe, although we can always define them pointwise, and that would ensure the resulting "sampling" looks much and much like the target function, one must understand infinite pairs would be required to specify $sin(x)$. Even with $\aleph_0$ points specified, there would be set of infinite functions who are not $sin(x)$ but just gives off the exact same value at all those specific points. This has a name, called pointwise convergence.

Outside those fixed set of points the family of functions can take arbitrary values - and thus specification via point pairs arguably pose a problem.

Luckily, for software we can do much better, which is the topic for the next section.

Software

A software is defined to be a Computable Function - mapping abstract vector space of input to the output vector space. The notion of using vectors is due to all real software works with many inputs and hence the state space is multidimensional which is the exact same space as output.

$$S : \hat{I} \to \hat{O}$$

where $\hat{I} :=< x_i >$ is the input vector while $\hat{O} :=< y_j >$ is the output vector. These vectors are defined not in physics sense, but pure mathematical sense. The only change between the pointwise defined function vs specified software is about being "Computable".

Software Test

A "Software" test is defined as a higher order function :

$$T : t < \hat{I}_t, S_t, \hat{O}_e > \rightarrow S_t(\hat{I}_t) = \hat{O}_t$$

In plain English, a test is comprise of Input vector $\hat{I}_t$, the software under test S_t, and the expected output vector $\hat{O}_t$, it runs the S_t with the input, and checks whether or not the expected output $\hat{O}_t$ matches against the actual output of the system :

$$S_t(\hat{I}_t) = \hat{O}_t$$

and it simply checks whether or not $\hat{O}_t = \hat{O}_e$, hence the range of the test is Boolean.

A software test, then contains a point specification for the desired Software.

A software test does not need to be computable. Unfortunately, any automated test, by definition needs to be computable. This also pose a problem for testing in general. A test that might not be computable is a human reporting software has hung or went into infinite loop. This is impossible to do algorithmically, unless we bound the time. This comes under Oracles in computation.

Branches and Partitions

Given software is written essentially using arithmetic logic and then conditional jump - this being the very definition of Turing Complete languages - that conditional jump ensures that the different inputs takes different code paths. A code path is a path (even having cycle) in the control flow graph of the software which starts at the top layer of the directed graph that is the code and ends in the output or bottom later.

Formally we can always create a single input node and output node in any control flow graph.

Treating multiple iterations of the same cycle as a single cycle, we can evidently say given the nodes of the graph is finite, there would be finite (but incredibly high) number of flow paths in the graph.

At this point we introduce the notion of equivalence class of input vectors to software. If two inputs $\hat{I}_x$ and $\hat{I}_y$ takes the same path P in the control flow graph, then they are equivalent.

This has immense implication in testing. Because this induces an equivalence partitioning on the input space itself, because all $\hat{I}_x$ in the same equivalence class can be treated as exactly equivalent, because all of them would follow the exact same code path in the control flow graph.

This effectively means by isolating all equivalence partitions and choosing one input member from each of them we can test the system the most optimal way. For example, if there are A, B, C, D equivalent classes, then choosing $\hat{I}_A \in A$, only one would test the code path for A, similarly for the rest. So instead of infinite inputs, only 4 inputs would suffice.

This brings the problem, formally to finding the equivalent classes.

That is impossible without the implementation. It is absolutely wrong to perceive that this technique is a specification driven one alone. This is a gray box testing it requires assuming some implementation details. The wikipedia article on this demonstrates this nicely.

But just how many equivalent classes would be possible? This definitely depends on the number of the conditional jumps. It is easy to prove that if there are B branches, then the bound of the equivalence class is $O(2^B)$ - a huge number. This also would be very important for a pragmatic discussion later.

The Equivalent classes would be called EQCP from now on because they partition the input set into Equivalent Classes.

Coupling in Software

Given individual EQCP are nothing but depicting paths in the control flow graph (CFG), the coupling said to exists between EQCPs E_x with path P_x and E_y with path P_y if and only if $P_x \cap P_y \neq \emptyset$.

That is, if paths P_x, P_y has some common edges, then E_x, E_y are coupled. In fact we can define the amount of coupling using similarity measures now, most easy one would be a Jaccard distance like measure:

$$C(E_x, E_y) = \frac{|P_x \cap P_y|}{|P_x \cup P_y|}$$

This essentially says - "Measure of the coupling between two equivalent classes is the amount of code flow shared between them relative to all code they have".

Test Driven Development as Equivalent Class Specification

By following the definitions till now, we can now formally define a software system specification in a finite, and correct way. If we can just specify the equivalence classes, then we can just fixed the software at those specification points and the resulting tests precisely, and correctly defines the software behavior. This must be taken as the formal definition of TDD.

> Given an abstract (not written) Software S_a, let's imagine the equivalence classes E_x such that E_x, E_y are independent and specify the input and output expected from each equivalence classes.

This system is provably complete and correct, by construction. Every test just ensures all individual EQCP behavior is passed via construction.

Correctness of TDD

The correctness of TDD for a practical application hinges on the following :

1. Is the specification complete enough (to take care of all the equivalent classes)?
2. Is the specification non contradictory ?

That it is impossible to get (1,2) done together follows from Godel's Incompleteness theorems, but that is applicable to any specification, not only Software. Thus this argument should not be admissible as failure of TDD in itself.

Now we ignore the notion of contradiction and focus on completeness and stability.

Completeness of TDD Spec

The business specification should be such that the formal specification of all possible Equivalence classes must be drawn from it. As it is bounded by $O(2^B)$ - this itself is not remotely possible. To understand how this bound works, a simple program unix `cat` has more than 60 branches. The equivalent class specification of this program is bounded by 2^{60} and the total electrons in the universe is 10^{80} form comparison.

But this again does not disprove the crux of TDD, it only points to the fact that formal EQCP is a practical pragmatic challenge.

Now an incremental TDD is when we add more specification to the mix of already existing ones. This incremental TDD is what we discuss in the next sections as this is the one which proponents of TDD talks about.

Analysis of Incremental TDD

Development in TDD

Note that the methodology does not specify how to implement the paths of each equivalent classes in the code. Hence evidently there is no way it can ever improve on the "non correct aspect of quality" of software, e.g coupling. In fact if not controlled this would bring in way more coupling than it was required due to application of other principles like DRY. There can be of course infinite way to conform to the "point wise convergence". These are some of the interesting properties of the methodology.

The very best non coupled way to deliver would be no equivalence class share any code path. This would solve the coupling problem, but code would be massively bloated. Any other way would reduce the code but ensure the classes would be coupled tighter.

Stability of TDD Spec over Iteration

It is at this point we go to the heart of the matter. Suppose, there is already an existing system in place with tests done the right way, at least catching up to 1% or 2% of the equivalence classes. Ideally we would want to have way more coverage, and that will be discussed later.

Is it possible to add more specification w/o rewriting existing equivalent classes?

The answer to this is key to the prospect of TDD.

Formally, Software S_r, as the equivalent classes $E_x \in \mathbb{E}_r$, and now more specification change is happening. The following questions need to be asked:

1. How many of the existing EQCP will not be effected by this?
2. How many new EQCP needs to be added?
3. How many EQCP needs to be removed?

As one can surmise, this is a fixed point iteration on the abstract space of the EQCP itself which we shall formally define later on.

Loose Coupling - A Life Saver

The answer to all these questions is coupling and coupling alone. If the implementation of those equivalent class was such a way that there was minimal coupling, then possibly less classes would be impacted via this. But this is not a principle of TDD in the first place in any form in any practical application of software development. In fact software principle like DRY would mandate code sharing, and hence there would be some coupling.

Pent Up Branching

The answer to the question [2] is in isolation if there would be K branches to implement the delta specification - new feature then, the isolated equivalent classes would be in $O(2^K)$, thus, the minimum new classes needed would be bounded by this value.

This chance may impact every possible tests because at most it can impact every equivalence class and at least it adds $O(2^K)$ classes and hence tests.

Incremental TDD - Chaos in EQCP space

Hence, in Incremental TDD, this culminates into a lot of those equivalence classes being thrown out, new classes being created - a fixed point iteration on the abstract space of the EQCP itself, which we can formally define as follows:

$$\mathbb{E}_{n+1} = \tau(\mathbb{E}_n, \delta_n)$$

Where at step n, $\mathbb{E}_n$ is the current set of EQCPs, while based on new specification (δ_n) and the $\mathbb{E}_n$ TDD τ produces new set of EQCPs ($\mathbb{E}_{n+1}$) for the next step $n+1$. This is the fixed point iteration of incremental software development from point pair specification or incremental TDD.

It is obvious that the first ever specification was :

$$\mathbb{E}_1 = \tau(\emptyset, \delta_0)$$

This is a fixed point iteration on the state space of EQCP, and this now depicts a dynamical, complex system.

The entire software for all practical purposes may go for entire rewrite - all existing test cases would need restructuring and rewrite. And this is equivalent to the chaotic condition in complex system - a very small input change (δ_n) would produce unpredictable EQCP changes of the system.

In the terms of complex dynamical systems - coupling in code results in "mixing" in the state space for EQCPs - $\mathbb{E}_n$. The larger coupling the system has, the more "mixed" the EQCPs are with one another. And this coupling changes on every iteration.

Is there a guarantee that chaotic patterns would emerge? Fairly regularly. Chaos in software development has been discussed about although not in much formal details like this.

We can try avoiding all of these by either:

1. Making the specification more relaxed - at that point it would specify almost nothing and there would be almost no chaotic behavior because of the state space of EQCP being reduced drastically. This is the placebo, the application of TDD w/o any formalism.
2. Or, we can try to decrease coupling, in which case it would bloat the software by not having shared code path - this would result is unimaginable bloat in the software - given we are looking at very large dimension of EQCP state space.

Evidently, then via [2] incremental TDD, therefore, can only be effectively done in practice when the $\mathbb{E}_n$ space is extremely small and the context of "Software" is very narrow.

Analysis of Behavior of Software Under TDD

Domain of Stability for Incremental TDD

What is so wrong about chaos appearing in the state space of the EQCP? Well, that means there is a lot of churn in terms of the changes - in the EQCP and that means - a lot of churns in the "pair points specifications" e.g tests which were to "hold the correctness of the software". Evidently this is a huge problem, if in one iteration which was created by a tiny change in specification impacts 50% of the test cases.

The chaotic thesis suggests that not this is only possible, but also highly likely due to the mixing of EQCPs in terms of coupling, and a direct result of code refactoring trying to apply DRY principle.

Hence the formal idea of just fixing input output points can not work generally unless we keep on reducing the scope of the specification. It is guaranteed to work at the Unit Test level by definition. Unfortunately the proponents of TDD want to make it work even at user specification level - where it entirely lose its rigor and has no provable applicability to either improve the quality of the product or the code itself.

The iteration over all existing equivalent classes and new ones makes it a fixed point iteration, and specifically if lucky it may be in the domain of stability, but such zones would be extremely narrow due to the large dimensions in which the system operates. This is well known in the domain of the complex system. It is what the original document wanted to talk about via Hill Climbing, which is not a terrible way of looking at TDD trying to slowly converging into the "Ideal final state of the software" at a step n. But this document does not want to take that route at all, because it is immaterial for now, for theoretical purpose, whether optimum was reached or not, we are only focused on producing "formally provably correct" Software as that is the superpower of formal TDD when done right as shown in this document.

Placebo view of Popular "Business Specification based" TDD

The previous issues culminates into less and less specific specifications used in the industry. At that point that they cover so less equivalence classes that TDD would lose all it's effectiveness which is to be found rigorously at the unit test level. Thus we do have a problem, if we specify more and more, the resulting software has high coupling thereby ensuring the iterations are destabilized. If we specify less and less the resulting diluted TDD is just homeopathy, water in the name of medicine but peoples believe making it "work" - a placebo.

Interestingly this is the most popular TDD in the industry because the only way non Unit Testing TDD can be done is by just showing they people are doing TDD, which is not even wrong. There is no effectiveness of TDD for anything that goes of unit test, as was proven. It gives a lot of people something to talk about and mental peace just like Homeopathy sans effectiveness other than placebo.

Proven Narrow Context Of Applicability

Not all is lost however. As it is proven, if we can go narrower and narrower, to the point when EQCPs stop sharing code with one another, TDD not only becomes formally correct, but also the ONLY methodology known to mankind to develop software. Do these exist? They do, these are the unit tests. We can now formally define scope for formal incremental TDD, which is guaranteed to work - e.g. create formal verifiable correct software as follows:

> Unit like tests where implementation of such features do not share any source code, e.g. Independent (completely decoupled) - such that in every iteration the decoupling holds true guarantee to hold to verifiable correct behavior.

And it is in this context TDD reigns supreme, not anywhere else. Of course one can try to use a scalpel to dig a canal, it just won't work. Any effort of using the scalpel to create a canal is not only misguided, but futile, and not even wrong.

Do iterative TDD, just ensure all EQCPs are completely decoupled - that is the theory. Now, in practice it is hard to do, even for Unit tests, so a small amount coupling should not really harm the effectiveness via that much - but at that point Chaotic behavior stems in.

Principles like AHA, WET comes in extremely handy in this regard. Even with very less coupling there is no absolute guarantee of formal correctness, but at least we are in the right track more or less unlike placebo.

Summary

Formal TDD, as presented here, is proved to produce correct software code. The issue with such production is of practical consideration, because inherently it increases coupling when done correctly to ensure we can further add more features to the existing software. If that is not the case, then the addition of more equivalence classes can would would rewrite the entire EQCP mapping by ensuring one must rewrite a very significant amount of tests, and this is also seen in reality. Anything at higher level it would just fail to work.

We also formally represented many informal concepts like "Coupling" and defined formally what "Unit" in unit testing supposed to mean.

TDD is not effective for user level testing, it is proved not to be because it lose out its correctness at that level effectively becoming at best Placebo.

Hence we propose TDD is to be done at the Unit Testing level only, where it works correctly and satisfactorily because of Units should be essentially maximally decoupled. And this should come as no surprise that is precisely what Kent Beck "rediscovered" while talking to other developers - "the only way to do unit like testing in Software".

References

1. Aleph Numbers : https://en.wikipedia.org/wiki/Aleph_number
2. Point Convergence : https://en.wikipedia.org/wiki/Pointwise_convergence
3. Test Vector : https://en.wikipedia.org/wiki/Test_vector
4. Computability : https://en.wikipedia.org/wiki/Computability
5. Oracle Machines : https://en.wikipedia.org/wiki/Oracle_machine
6. Higher Order Function : https://en.wikipedia.org/wiki/Higher-order_function
7. Decidability : https://en.wikipedia.org/wiki/Decidability_(logic)
8. Turing Completeness : https://en.wikipedia.org/wiki/Turing_completeness
9. Control Flow Graph : https://en.wikipedia.org/wiki/Control-flow_graph
10. Jaccard Distance : https://en.wikipedia.org/wiki/Jaccard_index
11. Equivalent Class : https://en.wikipedia.org/wiki/Equivalence_partitioning
12. Gray Box Testing : https://en.wikipedia.org/wiki/Gray-box_testing
13. Test Vectors Considered Harmful : http://fate.tttc-events.org/proceedings/5.1.pdf
14. `cat` source code : https://github.com/coreutils/coreutils/blob/master/src/cat.c
15. Coupling : https://en.wikipedia.org/wiki/Coupling_(computer_programming)
16. DRY : https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
17. Godel and System Specification : https://en.wikipedia.org/wiki/Gödel%27s_incompleteness_theorems
18. TDD As It was Intend to be : https://en.wikipedia.org/wiki/Test-driven_development
19. Complex Systems : https://en.wikipedia.org/wiki/Complex_system
20. Dynamical Systems : https://en.wikipedia.org/wiki/Dynamical_system
21. Fixed Point Iterations : https://en.wikipedia.org/wiki/Fixed-point_iteration
22. Chaotic System : https://en.wikipedia.org/wiki/Chaos_theory
23. Mixing : https://en.wikipedia.org/wiki/Mixing_(mathematics)
24. Software Development shows Chaotic Behavior : https://timross.wordpress.com/2010/01/17/software-development-and-chaos-theory/
25. Homeopathy - Delusion : https://en.wikipedia.org/wiki/Homeopathy_and_Its_Kindred_Delusions
26. Not Even Wrong : https://en.wikipedia.org/wiki/Not_even_wrong
27. Placebo : https://en.wikipedia.org/wiki/Placebo