A FORMAL ANALYSIS OF ITERATED TDD

HEMIL RUPAREL AND NABARUN MONDAL

Abstract. In this paper we formally analyze the software methodology called (iterated) Test Driven Development (TDD). We formally define Specification, Software, Testing, Equivalence Partitions, Coupling, to argue about the nature of the software development in terms of TDD. We formalize Iterative TDD and find a context in which iterated TDD "provably produce" "provably correct code" from "specifications" while being stable in terms of iterated code churns. We demonstrate that outside this context iterated TDD will exhibit chaotic behavior, implying unpredictable messy amount of code churn. We argue that the research finding of "ineffective" iterated TDD found by earlier researches are due to missing this context, while the findings of "effective" iterated TDD is due to accidentally falling into the context or simply placebo.

1. Canonical Definition of Iterated TDD

1.1. **Canon Definition.** We define TDD [1] as it is written in the canon article taken as the "Definition of TDD" [2] :

(1) Write a list of the test scenarios you want to cover
(2) Turn exactly one item on the list into an actual, concrete, runnable test
(3) Change the code to make the test (& all previous tests) pass (adding items to the list as you discover them)
(4) Optionally refactor to improve the implementation design
(5) Until the list is empty, go back to [2].

1.2. **Narrative.** The previous definition does not talk about any formal goals for iterative TDD. Hence, we formalize the objective of TDD as follows:

2010 *Mathematics Subject Classification.* Primary 68N30 ; Secondary 37B99, 68Q99, 93C99, 93D99, 68Q30.

Key words and phrases. Software ; Testing ; Test Driven Development; Formal Specification; Equivalence Class Partitioning; Dynamical Systems ; Chaotic Dynamics; System Stability; Lyapunov Exponent .

Hemil Ruparel : Dedicated to my parents and family without their presence we are nothing.

Nabarun Mondal : Dedicated to my late professor Dr. Prashanta Kumar Nandi.
Dedicated to my parents.
In Memory of : Dhrubajyoti Ghosh. Dear Dhru, rest in peace.

To ensure that we end up having a formally verifiable software in each step and in the end when all the "scenarios" are exhausted. Another "optional" objective is given as to "Improve the implementation design".

Note that it is not defined anywhere between one implementation to other what can be "improvement".

This is not a good starting point to formally analyze the methodology, as success metrics are not possible to be created on top of it. It is very imprecise, and open to interpretations.

In this paper we propose a formal methodology and provably demonstrate how "provably correct software" can emerge with clear metric of "amount of code churn" was done to attain it over the iterations - albeit in a very narrow context.

This practice we shall call formal Iterated TDD. We are calling the "canon" practice followed in the industry as "Iterated TDD" for reasons which would be apparent shortly.

2. Definitions

We would need some definitions to formalize the (Iterated) TDD pseudo algorithm.

2.1. Specification of Functions via Point Pairs.

Any function, computable or not, can be imagined to be pairs (potentially $\aleph_1$ [3]) 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 the function it is trying to describe.

2.1.1. *Consistency.*

A function is formally defined as a relation where it is impossible to have "re-mapping" e.g. same input point mapped to two different output points. The set of pair points must not have such spurious points, this we shall call consistency criterion. This will become a key point in case of software specification.

2.1.2. *Completeness.*

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 $f(x) = x$, identify function, but one can not define this function by keeping on 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 family of 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 [4].

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.

2.2. **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" [5].

2.3. **Software Test.** A "Software" test is defined as a higher order function [6] :

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

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 single point specification for the desired Software, this is the test vector [7].

A software test does not need to be computable in general. Unfortunately, any automated test, by definition needs to be computable. This also pose a problem for testing in general. Example of a test that is not computable [8] [9] can be : a human reporting software has hung or went into infinite loop. This is impossible to do algorithmically, unless we bound the time. This sort of scenarios comes under Oracles in computation [10].

2.4. **Code : Control Flow Graph, Branches.** Software is written essentially using arithmetic logic and then conditional jump - this being the very definition of Turing Complete languages [11]. This structure with 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 [12] (CFG) 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.

2.5. **Partitions : Equivalence Classes.** 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 and finding tests. 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 [13] in the control flow graph. There is another related concept called boundary value analysis (BVA) [14], but we would not go there, because that is not going to alter the subsequent analysis in any significant way.

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 - by restricting the number of "Software Test"s, as well as providing a full "coverage" in terms of specification.

For example, if there are A, B, C, D equivalent classes [15], 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. Notice that these are the most optimal set of inputs, the bare minimum to ensure that the system works in a provably correct manner.

This formally brings the problem to finding the exhaustive set of equivalent classes (let's call it $\mathbb{E}$) that completely describes one implementation of a "Software" system.

That is impossible without the implementation. It is wrong to perceive that this technique is driven by specification alone. EQCP is a gray box testing [16] technique as it requires assuming some implementation details [17].

What would be an upper bound of the number of such equivalent classes ? This depends on the number of the conditional jumps. It is easy to prove that if there are B branches, then the bound for the number of the equivalence class is $O(2^B)$ where $O(.)$ is "Big-Oh" one of the Bachmann Landau asymptotic notations [18], 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. There would be many EQCP for individual "features" in "Software".

2.6. **Coupling in Software.** At this point we introduce the phenomenon of coupling [19] between Equivalent Classes, when seen with respect to code implementation.

Given individual EQCP are depicting unique paths in the control flow graph (CFG), then coupling said to exists between EQCPs E_x with path P_x and E_y with path P_y if and only if $P_x \cap P_y \neq \varnothing$.

That is, if paths [13] P_x, P_y has some common nodes, 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 [20] like measure:

$$C(E_x, E_y) = \frac{|P_x \cap P_y|}{|P_x \cup P_y|} \tag{2.1}$$

This essentially says - "Measure of the coupling between two equivalent classes is the amount of code shared between them relative to all the unique code path they have together". We need to understand that even code shared for good reason, like applying DRY [21] and not doing it even methodically also would create coupling via this definition. Any shared function between two EQCP would mean coupling exists. As we shall see Coupling becomes a key phenomenon while analyzing the stability of software under Iterative TDD.

2.7. **Test Driven Development as Equivalent Class Specification.** We can now formally define a software system specification in a finite, and provably correct way.

If we can just specify the equivalence classes, then we can just fix the software output at those specification points and the resulting tests precisely, and correctly defines the software behavior. This must be taken as the formal definition of (non iterative, formal) TDD with absolute minimal test inputs:

> 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. Now, ensure all of these tests pass by writing the implementation.

This system is provably complete and correct, by construction. Every test just ensures all individual EQCP behavior is passed via construction. Given that was the entire specification, this means the system passes all criterion for the specification, and thus becomes provably correct.

The input output specifications can be immediately translated into tests, and that gives the formal provable meaning to TDD. Any random tests on features won't do, it have to be (at bare minimum) spanning the entire EQCP (the formal specification points).

This is the real superpower of TDD, formal verification baked into development. Although, truth to be told, this way of constructing software has been known for many decades.

And this is why the canonical TDD was called out as "iterated TDD" because this formal non iterative TDD model does not include change of specification, thereby does not follow any iteration and thus does not consider code churn thereof. This formal non iterated model is one single shot transformation of bunch of specifications points into code via transforming them into EQCP.

2.8. **Practical 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 [22], 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 when one tests gets added at one time (iterated or incrementally changed specification TDD).

2.9. **Practical 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 [23]. The equivalent class specification of this program is bounded by 2^{60} and the total stars in the universe are estimated to be 2×10^{24} for comparison.

But this huge numbers does not disprove the crux of TDD, it only points to the fact that formal EQCP is a practical challenge and to be handled pragmatically, probably via reducing the specification scope further and further.

3. Analysis of Iterated TDD

3.1. **Development under 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, one of them would be to lower 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. Because there are infinite way to conform to the "point wise convergence" but then the methodology does not specify any family of approach to do so. These are some of the key open problems of the methodology as it formally stands as of now.

A trivial non coupled way to construct code would be such that 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 to some extent.

This is a choice. We want to simultaneously minimize two metrics:

$$C_S = \sum_{x \neq y} C(E_x, E_y) \tag{3.1}$$

along with:

$$S_S = min_n\{K(S_n)\} \tag{3.2}$$

where S_S stands for "source code size" where $K(S_n)$ defines the optimal code size of the System S at n'th implementation trial. This is a very hard problem as Chaitin Solomonoff Kolmogorov Complexity (CSK) [24] is **non Computable** [5]. We do not even know if such a problem can be solved in formal setting. We posit it as an open problem in Software Development.

In lieu of that we continue in our analysis where we imagine a bit of necessary code coupling and try to reduce the code churn in terms of EQCPs. This coupling would have implication in iterated TDD, and we show a provable methodology that can reduce code churn in the later sections.

3.2. **Iterated TDD.** An Iterated (incremental) TDD is when we add more specification to the mix of already existing ones one step at a time under practical setting. This **incrementally added test based iterative TDD methodology** is what we discuss in the next sections as this is the one which proponents of TDD talks about. We note down it is different from the formal TDD we have established before - canon TDD is an iterated version of the formal TDD with specifications being added per iteration.

3.3. **Stability of EQCP under Iterated TDD.** Suppose, there is already an existing system in place with tests done the right way - following the EQCP method discussed earlier, e.g. following TDD.

Is it possible to add more specification w/o rewriting existing equivalent classes in a stable manner?

The sort of stability we are looking for is called BIBO Bounded input Bounded Output stability [25], that is, for a small change in specification, not much change would happen in the EQCP space.

This is the iterative TDD, applying this again and again. The answer to this is key to the prospect of iterative TDD.

Formally, Software S_r, has the equivalent classes $E_x \in \mathbb{E}_r$, and now more specification augmentation 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 the transformation step of a fixed point iteration on the abstract space of the EQCP. We shall get back to it slightly later.

3.4. **Additional 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.

At most it can impact every equivalence class and at least it adds $O(2^K)$ classes and hence tests. So, at the best case scenario, the total branches would become $O(2^B+2^K) = O(2^B)$ given $B >> K$. The complexity increases, but not drastically, unless $B = O(K)$.

3.5. **Impact of Coupling.** What happens when there is coupling? Instead of adding the terms, now because of dependency, the terms gets multiplied. Thus, with coupling the resulting complexity becomes $O(2^B \times 2^K) = O(2^{B+K})$. The delta change results in exponential growth even if $B \neq O(K)$.

This is a problem.

If the implementation of those equivalent class was such a way that there was minimal coupling, then less classes would be impacted via this step in the iteration. 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 and modular programming would mandate code sharing, and hence there would always be some coupling.

3.6. **Iterated TDD as a Dynamical System.** At this point we can formally represent iterated TDD as a dynamical system [26].

As discussed, this EQCP merging 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 now formally define as follows:

$$\mathbb{E}_{n+1} = \tau(\mathbb{E}_n, \delta_n) \tag{3.3}$$

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 system 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, iterated TDD.

It is obvious that the first ever specification was done with empty equivalent classes ($\mathbb{E}_0 = \varnothing$) and initial specification of δ_0:

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

This is how formally iterated or incremental TDD looks like. These equations now depicts a dynamical, complex system with am initial boundary value or starting condition.

3.7. **Stability Space.** While EQCP space is nice to visualize what is happening for real in terms of Software Specification and Test cases, it is not descriptive enough to translate into numbers so that we can track the trajectory of the Dynamical System.

How much change in the EQCP space is happening on each iteration of iterated TDD? It is impossible to comprehend that in the EQCP space.

For gaining this insight we would need a metric, that would define how stable the system is over the iterations in terms of retaining past EQCPs - how much code remained same between iterations.

We define the stability metric as follows :

$$\Sigma_{n+1} = 1 - \frac{|\mathbb{E}_n \cap \mathbb{E}_{n+1}|}{|\mathbb{E}_n \cup \mathbb{E}_{n+1}|} \; ; \; \Sigma_n \in \mathbb{Q} \cap (0,1) \tag{3.4}$$

The stability metric Σ also depicts a metric space [27] with distance between two stability points $a, b \in \Sigma$ as defined to be : $d(a,b) = |a - b|$.

3.7.1. *Stable Point : 0.* Observe the following, if we ensure that no EQCP has any shared code, then the only way to make change is to simply add new code, and thus $\mathbb{E}_n \subset \mathbb{E}_{n+1}$, and that gives minimum value of Σ if and only if $|\mathbb{E}_{n+1} \setminus \mathbb{E}_n|$ can be minimized .

A value of Σ close to 0 shows the system has been very stable between last to the current iteration. This is when "very loose" coupling ensured that we can create branches which do not interact with existing branches that much. We present order of magnitude estimates for "highly stable" uncoupled ${}^U\Sigma$ value as follows:

$${}^U\Sigma_{n+1} \approx 1 - \frac{|\mathbb{E}_n|}{|\mathbb{E}_{n+1}|} \approx 1 - \frac{O(2^B)}{O(2^B + 2^K)} \approx 1 - \frac{1}{1 + 2^{K-B}} \approx 0 \; ; \; B >> K \tag{3.5}$$

We note that it is impossible to reach value 0 under any circumstances other than when $\mathbb{E}_n = \mathbb{E}_{n+1}$ which means, the specification δ_n did not change anything in EQCP space, e.g. a complete dud or spurious specification.

Importantly, there can be cases where even without coupling, as demonstrated by : $|\mathbb{E}_n| << |\mathbb{E}_{n+1}|$, then even though $\mathbb{E}_n \subset \mathbb{E}_{n+1}$, the stability would be going for a toss - this is driven by having $B = O(K)$.

3.7.2. *Unstable Point : 1.* Now the other side of the coin is when $\mathbb{E}_n \cap \mathbb{E}_{n+1} \approx \varnothing$, in this case the value of Σ goes to 1.

A value of Σ close to 1 shows the system has been very unstable between last to the current iteration. This is when "strong" coupling ensured that we need to rewrite a lot of the EQCP implementations in code.

The "reasonably coupled" ${}^C\Sigma$ estimate would be as follows:

$${}^C\Sigma_{n+1} \approx 1 - \frac{O(|\mathbb{E}_n \cap \mathbb{E}_{n+1}|)}{O(|\mathbb{E}_n \cup \mathbb{E}_{n+1}|)} \approx 1 - \frac{O(2^B)}{O(2^{B+K})} \approx 1 - \frac{1}{2^K} \approx 1 \; ; \; K >> 1 \tag{3.6}$$

Where K is some constant estimating the branch changes due to δ as depicted in previous section.

3.8. **Guiding Stability Algorithm.** Assuming coupling would almost always be present, one way for us to avoid unpredictable jumps in the stability, we can device our development strategy such that the Σ does not change drastically towards 1.

At this point, if there were many alternative way to program (P_i) the δ_n change, we may want to chose the alternative P_x way to program which minimizes Σ_{n+1}. If we do, then the system remains stable in the short term. But this is a direct anti thesis of "less code change and faster changing ability", as it minimizing Σ_{n+1} culminate into more code change, because it would inherently try to lose some coupling!

More importantly, this computation of minimizing the Σ post applying the δ change can be greedy, but it is evident that here is where hill climbing creeps up, there can be a minima hidden somewhere else.

At this point, in the worst case it would boil down to applying all specification changes $\{\delta_i\}$ which would have have a factorial runtime or, would be in NP. This is anti agile, and definitely not "small incremental change", this is a lot of change, pre-computed, and applied to minimize code churn.

By this time, we have understood that practically following guided stability is already very hard, however, worse is yet to be seen by us. Unfortunately even with this guided approach there would be some problems which would not go away, in the long term, that is the discussion of the next section.

3.9. **Chaos in Stability space.** We now proceed to demonstrate that the iteration driven by (τ, δ_n) in Stability Space Σ has characteristics of a system capable of showcasing chaotic dynamical behavior [28].

Given there is no universally agreed definition of chaos - we - like most people would accept the following working definition [29] [30]:

> Chaos is aperiodic time-asymptotic behavior in a deterministic system which exhibits sensitive dependence on initial conditions.

These characteristics would now be demonstrated for iterated TDD.

(1) ***Aperiodic time-asymptotic behavior*** : this implies the existence of phase-space trajectories which do not settle down to fixed points or periodic orbits. For practical reasons, we insist that these trajectories are not too rare. We also require the trajectories to be *bounded*: *i.e.*, they should not go off to infinity.

    The sequence $\Sigma_n \in \mathbb{Q} \cap (0, 1)$ is bounded by definition. The trajectories are not rare, and it is practically impossible for the sequence to settle down to periodic orbits or converging sequence. Note that w/o the presence of coupling this sequence can be made to orbit around approximating 0 most of the time.

(2) ***Deterministic*** : this implies that the equations of motion of the system possess no random inputs. In other words, the irregular behavior of the system arises from non-linear dynamics and not from noisy driving forces.

One can argue that the sequence is driven by δ_n - an external input, but it is not. Iterative TDD has this baked in, as part of the system iteration description , and the processing of it is algorithmic in the formal methodology which we present for formal correctness for the software. In fact we can argue that the sequence δ_n can be specified beforehand, and it would make it fully deterministic and it would not impact our analysis.

(3) ***Sensitive dependence on initial conditions*** : this implies that nearby points can be spread further over time while distant points can come close over time - e.g. stretching and folding of the space. In fact it is said to be:

> Chaos can be understood as a dynamical process in which microscopic information hidden in the details of a system's state is dug out and expanded to a macroscopically visible scale (*stretching*), while the macroscopic information visible in the current system's state is continuously discarded (*folding*). The system has a positive Lyapunov exponent [31].

This is evident in case of coupling.

CFG comprise of the micro details which culminates into the the space of EQCP, and merging further specification over that produce the sequence Σ_n. Inherently a lot of micro details are being pushed into visibility and then again being discarded as in the Σ space, the information about current complexity of the system (EQCP space $\mathbb{E}$) does not exist.

We shall now proceed to formally demonstrate that Lyapunov exponent is positive for Σ.

Given two nearby points in Σ_n , say $a, b : |a-b| < \epsilon$, there is no guarantee that in next iteration how further apart the sequence would go, given even exactly same specification of δ_n . Let $\Sigma(p, \delta)$ be the next iteration sequence after starting from p in Σ_n post applying the same specification change δ.

Then $|\Sigma(a, \delta) - \Sigma(b, \delta)| \neq 0$ holds true almost always for all practical purposes.

Let us define the function $\Delta(a, b, \delta)$ as follows:

$$\Delta(a, b, \delta) = \frac{|\Sigma(a, \delta) - \Sigma(b, \delta)|}{|a - b|} \tag{3.7}$$

Then, a **stretch** happens when $\Delta(x, y, \delta) > 1$ and a **fold** happens when $\Delta(x, y, \delta) < 1$.

This is to say, stretch increases the distance between the trajectories starting with (a, b) while fold reduces it. We notice that the definition of Lyapunov exponent of the Σ would be as follows:

$$\lambda = ln(\Delta(a, b, \delta)) \tag{3.8}$$

We can approximate $\Sigma(x, \delta)$ in presence of some coupling - where B_x is the branching at x and K_x is the addition of branching due to application of δ as follows (estimating from previous section):

$$^{C}\Sigma(x, \delta) \approx 1 - \frac{O(2^{B_x})}{O(2^{B_x + K_x})} \approx 1 - \frac{1}{2^{K_x}}$$

This when substituted reduces to:

$$\Delta(a, b, \delta) \approx \frac{|\frac{1}{2^{K_a}} - \frac{1}{2^{K_b}}|}{|a - b|} \approx \frac{|2^{K_a} - 2^{K_b}|}{2^{K_a + K_b}|a - b|}$$

Now we choose a suitable ϵ for our purpose to simplify the expression as well as minimize it:

$$\epsilon < \frac{1}{2^{K_a + K_b}}$$

Thus making the smallest bound possible for Δ as :

$$\Delta(a, b, \delta) \approx |2^{K_a} - 2^{K_b}| \approx \theta(2^L) \ ; \ \forall(K_a \neq K_b) \ L > 1$$

And this immediately demonstrates that Lyapunov Exponent for the system is positive ($\lambda > 0$) :

$$\lambda = ln(\Delta(a, b, \delta)) \approx L \times ln(2) \ ; \ \forall(K_a \neq K_b) \ L > 1 \tag{3.9}$$

thereby proving that the Σ map is expansive and hence Chaotic under the influence of coupling.

We can argue the same in a semi formal way.

Evidently, if only folding happens, then every sequence would converge. This is an extreme view. In the same way if only stretching happens, then because the sequence is bound, it must converge again to 0 or 1. This is another extreme view.

We can safely say the probability that for every tuple (a, b, δ) that the $\lambda > 1$ would be 0. So goes the same for $\lambda < 1$.

It is much more plausible that a function like this would have some intervals where it would stretch and some intervals where it would fold depends on the δ. This is the most likely phenomenon which invariably would generate a sequences diverging and converging in Σ thereby producing the dynamic process that stretches and folds - and thus creating sensitive dependence on initial condition, the hallmark of chaos.

The above points make it very clear that the sequence Σ may show all properties of chaotic dynamics. Which proves that iteration of iterated TDD can and would show chaotic dynamics.

4. Practical Considerations for Software Development Under Iterative TDD

4.1. **Identifying Chaotic Trajectory.** Is there a guarantee that chaotic patterns would emerge on each case? No one knows. Chaos in software development [32] has been discussed about although not in much formal details like this. If we are very lucky it would not, but it is hard to tell. Only by carefully monitoring the sequences we would be able to claim whether we entered any chaotic sequence or not and this formalism gives a metric such that the sequence can be tested for emergence of chaos - by following Kantz [33]. That would be the empirical way of measuring on each iteration how the progress is happening. Given agility is the name of the game now, we can add 52 data points a year for each project if weekly shipping of software is followed.

4.2. **Domain of Stability for Iterated TDD.** Let's imagine the worst case, almost all of the sequences would be chaotic.

What is so problematic about chaotic dynamics appearing in the phase of "stability" of EQCP ? This means there might a unpredictable amount of churn in terms of the changes in the EQCP. And that means churns in the "pair points specifications" e.g tests which were to "hold the correctness of the software", implying a unpredictable, possibly a very high implementation change MUST happen.

If in one iteration which was created by a tiny change in specification impacted 50% of the test cases to refactor source code and tests thoroughly, evidently this would become a huge problem.

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 and rapid, small iteration on specification can not work in general unless we keep on reducing the scope of the specification.

It is only guaranteed to work (produce provably correct software and predictable amount of code churn) at the lowest abstraction level if there are very less coupling by definition. Unfortunately the proponents of TDD want to make it work even at user specification level - where it entirely lose out its rigor and has no provable applicability to either improve the quality of the product or the code itself.

4.3. **Uncertainty Principle of Iterated TDD.** We have uncovered an uncertainty principle [34] of sorts here:

> With coupling at play, if we try to fix more specification by specifying more EQCP, then the code churn becomes unpredictable. And if we do not go exhaustive on EQCP, then the formal correctness software producing characteristics of the methodology disappears.

It seems in the presence of coupling, we can either choose formal correctness or choose code churn stability, not both.

This insight is unheard of, but the theory points us in this direction. If the chaotic thesis is correct, this is to be taken as a foundational law of Software Engineering.

While this demonstrates why coupling is a problem, however, this is much stronger thesis, this tantamount to **any shared code is a problem if the code supposed to change later**.

4.4. **Revisiting Guided Approach.** Readers may argue that how then this analysis does not apply to any other software development process? The answer lies in the guided approach. In case, if one does not make the software fixed via hard test driven specification, then there is loss of "correctness" - granted, but there is a lot of "wiggle" room to build the system.

With the guided approach one can even try to avoid the entire chaotic trajectories by prioritizing specifications or even rejecting it for the time being, till a suitable time comes to apply such that the stability is not changed that much.

This, evidently is what non agile waterfall, or iterated waterfall [35]was all about. In fact we are formally defining prototypical development at this point [36].

Would they avoid the unstable paths? Sometimes. But mostly they would make the system "slower" in the stability space. Here, we are not talking about the slowness of delivery, we are talking about slow movement of the system in the stability space. This way, it would take a very long time to reach a chaotic state.

4.5. **Path Forward - Approaches.** From the last section to avoid these chaotic sequences 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 a cargo cult approach, producing only 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] iterated 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.

4.6. **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 effectively sharing code with one another, TDD becomes formally correct, also the methodology to develop software in regular iteration with predictable churn. This narrow specification contexts are in fact the unit tests with very less coupling which guarantee of becoming chaos free!

We can now formally define scope for formal iterated TDD, which is guaranteed to work - e.g. create formal verifiable correct software as follows without ever destabilizing source code:

> 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. Anything other than that - correctness or stability can not be guaranteed. Just like 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, this, now becomes a formally correct software producing code churn wise stable methodology. 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 [21] comes in extremely handy in this regard. Even with very less coupling there is no absolute guarantee of code stability, due to emergence of chaos but at least we are in the right track by being formally correct, and the resulting chaos can be tamed.

4.7. **A Perspective on Popular "Business Specification based" TDD.** The previous issues culminates into less and less specific specifications used in the industry. At that point 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 [37].

This is not hard to understand, as TDD mandates writing the tests first, there are some tests, for sure, better than none, and this essentially ensures there is at least some correctness in the mix. The fear of failing tests ensures code is often correctly written. It has been well understood that developers tend to write better code just because there would be testers who would test it. This however does not consider the "cost" of stability in code churn. This metric, surprisingly was never studied!

Interestingly "Business Specification Driven TDD" is the most popular TDD in the industry. This "Some input,output are verified" is not really an effective methodology, given the nature of the number of tests required runs in exponential numbers in terms of the EQCP for the features.

However, it gives a lot of people something to talk about and mental peace just like Homeopathy sans effectiveness other than placebo as it was found out in another research : [38].

We can also safely say, any low level, low coupled, EQCP based formal TDD method would be reasonably successful, if those practices were to be followed, iterated TDD would definitely be very effective. There are some publications where it has been shown to do exactly that [37].

4.8. **Cargo Cult "Software" Engineering?** We can therefore conclude that iterated TDD without understanding the applicability context is like washing your hand with water before you eat, while the "washing hand" would be a good practice, but if the water used was filthy, it would degenerate to numerable problems. This is the status of industry with respect to TDD, for those who are into the right context, it works, give or take. Those who are not, it does not.

We conclude by making a much more starker remark, the proponents of TDD, or "industry best practices" stopped asking "is this effective or provable" a long time ago. Their new established position is : "No evidence required for common sense practices". In fact, this is the verbatim response when asked about efficacy and provability of some of the best practices:

> You want to debate seriously? Then you have to drop the ridiculous sense that "Good Practices" require scientific evidence before they can be realized to work - which would disprove much of the "Good Practices" which are "successfully used" in the industry.

Even if we ignore the irony of the previous quote, one but just wonder if evidently Software had become entirely cargo cult [39], the above quote proves it beyond doubt. Very few admit it openly, but it is what it has become.

5. Closing Remarks

Formal iterated TDD, as presented here, is shown to produce correct software code. The issue with such production requires a lot more formal and practical considerations.

When done correctly (by EQCP and reducing coupling between them) it ensures we can further add more features to the existing software while maintaining stability as well as correctness as we go.

If that reduction of coupling is not followed, then the addition of more equivalence classes could and most definitely would modify a significant amount EQCP mapping by ensuring one must rewrite a very significant amount of tests, as well as implementations. This is also seen in reality. Anything at any further higher level of abstraction that Unit like tests would have impact like placebo.

Hence we propose iterated 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 keeping an constant eye on the coupling generated by those tests being constantly added, which is hard, but not impossible to do and shows provable theoretical efficacy: provably correct software production along with predictable code churn.

References

[1] Various, "Test Driven Development." https://en.wikipedia.org/wiki/Test-driven_development, 2024. [Online; accessed 3-July-2024].

[2] K. Beck, "Canon TDD." https://tidyfirst.substack.com/p/canon-tdd, 2023. [Online; accessed 3-July-2024].

[3] "Aleph Numbers." https://en.wikipedia.org/wiki/Aleph_number, 2024. [Online; accessed 3-July-2024].

[4] "Pointwise Convergence." https://en.wikipedia.org/wiki/Pointwise_convergence, 2024. [Online; accessed 3-July-2024].

[5] "Computability." https://en.wikipedia.org/wiki/Computability, 2024. [Online; accessed 3-July-2024].

[6] "Higher Order Function." https://en.wikipedia.org/wiki/Higher-order_function, 2024. [Online; accessed 3-July-2024].

[7] "Test Vector." https://en.wikipedia.org/wiki/Test_vector, 2024. [Online; accessed 3-July-2024].

[8] "Decidability in Logic." https://en.wikipedia.org/wiki/Decidability_(logic), 2024. [Online; accessed 3-July-2024].

[9] "Halting Problem." https://en.wikipedia.org/wiki/Halting_problem, 2024. [Online; accessed 3-July-2024].

[10] "Oracle Machines." https://en.wikipedia.org/wiki/Oracle_machine, 2024. [Online; accessed 3-July-2024].

[11] "Turing Completeness." https://en.wikipedia.org/wiki/Turing_completeness, 2024. [Online; accessed 3-July-2024].

[12] "Control Flow Graph." https://en.wikipedia.org/wiki/Control-flow_graph, 2024. [Online; accessed 3-July-2024].

[13] "Path in Graph Theory ." https://en.wikipedia.org/wiki/Path_(graph_theory), 2024. [Online; accessed 3-July-2024].

[14] "Boundary Value Analysis." https://en.wikipedia.org/wiki/Boundary-value_analysis, 2024. [Online; accessed 3-July-2024].

[15] "Equivalent Classes." https://en.wikipedia.org/wiki/Equivalence_class, 2024. [Online; accessed 3-July-2024].

[16] "Gray Box Testing." https://en.wikipedia.org/wiki/Gray-box_testing, 2024. [Online; accessed 3-July-2024].

[17] "Equivalent Partitioning." https://en.wikipedia.org/wiki/Equivalence_partitioning, 2024. [Online; accessed 3-July-2024].

[18] "Big Oh Notation." https://en.wikipedia.org/wiki/Big_O_notation, 2024. [Online; accessed 3-July-2024].

[19] "Software Coupling." https://en.wikipedia.org/wiki/Coupling_(computer_programming), 2024. [Online; accessed 3-July-2024].

[20] "Jaccard Index." https://en.wikipedia.org/wiki/Jaccard_index, 2024. [Online; accessed 3-July-2024].

[21] "Do Not Repeat Yourself." https://en.wikipedia.org/wiki/Don%27t_repeat_yourself, 2024. [Online; accessed 3-July-2024].

[22] "Incompleteness Theorems." https://en.wikipedia.org/wiki/GÃűdel%27s_incompleteness_theorems, 2024. [Online; accessed 3-July-2024].
[23] "Cat Source ." https://github.com/coreutils/coreutils/blob/master/src/cat.c, 2024. [Online; accessed 3-July-2024].
[24] "Kolmogorov Complexity ." https://en.wikipedia.org/wiki/Kolmogorov_complexity, 2024. [Online; accessed 3-July-2024].
[25] "BIBO Stability." https://en.wikipedia.org/wiki/BIBO_stability, 2024. [Online; accessed 3-July-2024].
[26] "Dynamical System." https://en.wikipedia.org/wiki/Dynamical_system, 2024. [Online; accessed 3-July-2024].
[27] "Metric Space." https://en.wikipedia.org/wiki/Metric_space, 2024. [Online; accessed 3-July-2024].
[28] "Chaos Theory." https://en.wikipedia.org/wiki/Chaos_theory, 2024. [Online; accessed 3-July-2024].
[29] "Definition of Chaos ." https://farside.ph.utexas.edu/teaching/329/lectures/node57.html, 2024. [Online; accessed 3-July-2024].
[30] "Characteristics of Chaos." https://math.libretexts.org/Bookshelves/Scientific_Computing_Simula 2024. [Online; accessed 3-July-2024].
[31] "Lyapunov Exponent ." https://en.wikipedia.org/wiki/Lyapunov_exponent, 2024. [Online; accessed 3-July-2024].
[32] "Software Development and Chaos Theory." https://timross.wordpress.com/2010/01/17/software-dev 2010. [Online; accessed 3-July-2024].
[33] H. Kantz, "A robust method to estimate the maximal lyapunov exponent of a time series," *Physics Letters A*, vol. 185, no. 1, pp. 77–87, 1994.
[34] "Fourier Uncertainty Principle." https://en.wikipedia.org/wiki/Fourier_transform#Uncertainty_pri 2024. [Online; accessed 3-July-2024].
[35] "Iterative Development ." https://en.wikipedia.org/wiki/Iterative_and_incremental_development, 2024. [Online; accessed 3-July-2024].
[36] "Software Prototyping." https://en.wikipedia.org/wiki/Software_prototyping, 2024. [Online; accessed 3-July-2024].
[37] V. Bakhtiary, T. J. Gandomani, and A. Salajegheh, "The effectiveness of test-driven development approach on software projects: A multi-case study," *Bull. Electr. Eng. Inform.*, vol. 9, pp. 2030–2037, Oct. 2020.
[38] I. Karac and B. Turhan, "What do we (really) know about test-driven development?," *IEEE Software*, vol. 35, pp. 81–85, 07 2018.
[39] R. P. Feynman, "Cargo Cult Science ." https://calteches.library.caltech.edu/51/2/CargoCult.htm, 1974. [Online; accessed 3-July-2024].

Pune
Email address: hemilruparel2002@gmail.com

Hyderabad
Email address: nabarun.mondal@gmail.com