

Does your software do what it is supposed to? Hillel Wayne teaches TLA+

Mar 12, 2018 • Max Mautner • Guest: [Hillel Wayne](#) • 42 mins read

Today [Hillel Wayne](#) joined us to discuss TLA+ and *software verification methodology*:

- what is it?
- who uses it?
- should you learn it?

Enjoy!

Full text transcript below the fold:



Audio:

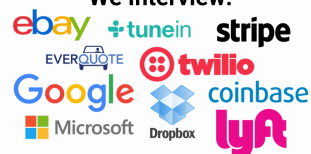


Get exclusive updates in the Accidental Engineer Newsletter:

Subscribe

Listen on Apple Podcasts

We Interview:



Show Notes:

- [Hillel Wayne's website](#)
- [@hillelogram](#)
- [Learn TDD](#)
- [Why TDD isn't crap](#)
- [Z notation](#)
- [Use of Formal Methods of AWS](#)
- [TLA+](#)
- [Learn TLA+](#)
- [Software Contracts](#)
- [Alloy](#)
- [Runway](#)
- [Graphviz](#)
- [Mermaid](#)
- [Property Based Testing](#)

Text Transcript

Max: Hillel Wayne has spoken widely on topics of software correctness, and the practices that professionals have adopted to improve how software is built on time and without bugs.

“How will you know your software does what you set out to do?”

That's the topic of our conversation today as Hillel shares with us some tools that exist for clarifying whether your algorithm will work before you start implementing it.

Welcome all. Max of the Accidental Engineer. Today we are joined by Hillel Wayne of Chicago, Illinois. Welcome Hillel!

Hillel: Hello. Thanks for having me on.

Max: Pleasure to have you. Today we're going to be talking a bit about one of Hillel's favorite subjects, software verification methods. Assuming layperson's knowledge of software, just a very general gist, do you mind introducing people to what software verification methods are Hillel?

Hillel: Sure. A lot of people in software use tests, where you say, “here is my program, here is my input, here’s the output I want, and running that.”

That’s the idea behind tests, automation tests from development et cetera. That’s a subfield of a broader field called “software verification” “or software correctness,” which encompasses all of the techniques we use to make sure our software doesn’t have bugs in it.

Max: This has a rich history—software that backs important aspects of our society, like aeronautics, NASA, health devices.

I want to illustrate for our audience just how important software verification methods are. Can you share a little bit about the history of the particular methods that you have used and which you find most interesting?

Hillel: The history of software verification’s actually pretty long. A lot of the work was done before we even had modern test suites. Probably the best place to start in terms of not going too far back, is a language called [Z](#), which I believe was done in the 1970s or 80s, where it was a way of specifying a program as an abstract blueprint.

This was used to develop rail systems. I believe the biggest success was a France public transit network.

Then there’s the other half to that. There was also a lot of work done with Pascal—what’s called stepwise refinement, taking ideas and turning them into code. That led to the idea of the [Liscov Substitution Principle](#). That created its own tangent, especially culminating in Ada and the [SPARK](#) use of formal verification.

Max: From what I understand there is some crossover from traditional fields of physical engineering, like mechanical, civil, et cetera—that crossed over to software when software and modern day computing was invented.

There’s various keywords that I have come across that have physically corollaries—like a test “rig” or a “mock” that have their predecessors in physical tests. Where maybe you would “mock” a component to a physical device as a form of a load test for a mechanical engineer.

I haven't really introduced how you got into software engineering! How did you get into software engineering?

Hillel: I was actually trained in undergraduate as a physics/mathematician. Got a physics BA and a mathematics degree from the University of Chicago and was originally planning to go to physics grad school, as an experimentalist.

That gave me a pretty big grounding in scientific method and scientific investigation et cetera. Then, in my last time in college, I realized that I much preferred the act of *doing* physics as the setting up the systems and the programs and running the experiments, versus the actual ideas in physics itself.

The more I investigated that, the more I realized that what I enjoyed most was the programming aspects and the handyman aspects. I figured that programming does a lot of really interesting stuff and it pays better than grad school so I decided to go into that.

Max: Yeah. Do you mind sharing about where you worked after college and what types of businesses they were?

Hillel: After I graduated college, I moved to San Francisco for about a year and a half and worked for EdModo, which is an education company. They do social media for schools. After that I realized that I didn't actually like San Francisco that much and moved back to Chicago.

Then I spent another two and a half years working for a small startup called eSpark, which does different educational software for schools. Now I'm currently on sabbatical for a bit.

Max: Educational software seems to have been a theme—what are some of the computing problems that were common to the two employers that you had in the industry?

Hillel: Full disclosure, at least with the first job, I was mostly starting as a junior engineer, so I was mostly doing product work and front end work. Not to say that only young junior engineers do front end or that front end is only for junior engineers, but that was where they spotted all their juniors to start out.

With eSpark, it turns out that a lot of the problems we had ended up being accidental complexity, where we had a product that was fundamentally an iPad app, which was really effective for teaching students. All of the support systems and development was what I ended up focusing on, and building a lot of my own experience, my own work.

For example, how do you get an app onto 500 students in a school that's running a really poorly run MDM that's running on a Mac Mini in a broom closet without crashing any of the iPads or the Mac Mini or the app itself.

Max: When you say accidental complexity, I know you've spoken at conferences, you have a very well-written blog that I myself read, about topics of reducing complexity or handling complexity in software.

Do you mind sharing anecdotally any of your personal experiences, trying to combat software complexity on the job?

Hillel: First I want to start out saying that "accidental complexity" probably was the wrong word there.

Most people divide it into essential complexity, which is things that are fundamentally hard, and accidental, which is things that are hard because bureaucracy and tooling and the work is hard.

I think one example of one of the more interesting systems we had to deal with is a lot of the stuff we worked with ending up being concurrent.

Things were going on at different times and over different timelines that didn't necessarily happen in one single order. Depending on the order, it could trigger bugs. One of the more interesting bugs we had to deal with was when some third party software where if you ended up setting too many commands over a short period of time, it would start deleting files at random from the local machines.

Max: That sounds terrific :)

Hillel: It was a challenge.

Max: What was the solution? How did you guys tackle it?

Hillel: In the beginning, we tackled it with a fairly—what’s the right word here? I don’t want to say “brutal,” but that’s the first word that comes to mind. A very strong-handed solution of guaranteeing they were only sending one command at a time even if we really needed to send more. We were just spacing them out.

It was a simple solution but we thought it worked. Then it turned out it didn’t work because we had made some mistakes in our own mocking and our own control.

In order to ultimately fix that, that led me to [TLA+](#) and formal verification. We ended up designing the system in abstract to make sure the bug was never triggered based on their description of it and then we implemented that.

Max: We should absolutely take a moment to plug [LearnTLA.com](#), a website that Hillel put together that is a crash course introduction for people who are curious about these abstract methods that Hillel’s mentioning right now.

You mentioned “in the abstract.” What is TLA as a programming language—is it intended to specifically write tests? What is the 20,000 foot overview?

Hillel: TLA+ isn’t a programming language—it’s what’s called a “specification language.”

The idea is that instead of writing a program, you write a *specification* of a program. The idea being that this is, as you said, a 20,000 foot blueprint of what you want the program to do.

You specify all the data structures in the abstract, the algorithms that it’s going to use. Then you can use it to verify that it has the properties you want.

In the ways that we used it [at eSpark], we specified the abstract system and we’d say, “never do anything that they said would trigger this one bug.” Then we used the model checker called TLC, which explored the entire state space of the design and could tell us whether or not any of the possible situations that could arise would trigger the bug.

Max: On the one hand, you’re describing the invariance of the system, the things that should never be violated. On the

other you're describing the state space that you want the model checker to explore, to verify that none of these invariance are violated. Is that a fair description?

Hillel: That's a fair description. It also reminds me of one of my favorite things about TLA+.

TLA+ stands for *Temporal Logic of Actions*. Not going to go into the theory here, that's what it stands for. The model checker, TLC, no one actually knows what it stands for. I've checked every single paper on the subject. I've checked all of the books written about TLA+ and I've asked the developers, and nobody remembers. It's just TLC.

Max: Fun trivia. An aside about trivia is that [Leslie Lamport](#), the author of LaTeX, is also the author of TLA+. Perhaps we could give him a ring and have him come on the show and do a three-person episode!

He might be able to tell us the origins of TLC, if I'm getting that right, that he was the author.

Hillel: I believe it was one of his grad students or post doc students who authored it. He actually said that he thought it was impossible to model check TLA+ when he first came up with it. It was just supposed to be a human design and then one of his students was like, I built the model checker. Yeah, we can test it now.

Max: I want to impress upon our audience that TLA+ has a pretty well-known use case which is that it's been used at Amazon Web Services in several of their biggest software engineering themes. S3, DynamoDB. For those people who don't know what those are, there's a few famous incidents from the last several years where Amazon's S3 services had downtime in which portions of the internet have been inaccessible.

You guys can imagine that software verification methods are extremely valuable for verifying that changes to services like S3 and DynamoDB, don't take down portions of the wider public internet, as used by websites like New York Times et cetera, et cetera. I want to impress upon our audience just how important tools like TLA+ might be at companies like Amazon. They found a use case for Hillel. Hillel, do you mind sharing about how you came across it

and how you ended up using it to solve this problem at eSpark?

Hillel: Sure. I ended up working on this bug and a few other bugs with similar systems that was proving very challenging. Normal testing methods weren't helping because it was so complicated and involved multiple third party systems. We couldn't use [inaudible 00:13:04] contracts because they were only for our stuff. It ended up being this really huge complicated problem that we really couldn't tackle. Around the same time I ended up switching my ADD medication, which meant I had about three days of obsessions about everything I encountered. I ran into this one person I know, Ron Pressler, who's another big figure in the TLA+ community. Check out his work, he does a lot about the theory on it. He ended up linking the case study that you mentioned, the use of formal methods at Amazon Web Services, I read that and thought, this looks really cool. I spent three days doing nothing but studying it. We ended up trying it at eSparks to see if it could help. It took about an afternoon to spec out the system. They didn't really tell us about three major bugs that we had no idea could possibly happen, based on our design.

Max: One of the distinctions I want to make ... First off, we'll include a link to this Amazon paper in the show notes.

Hillel: I'm going to send you so many links for the show now :)

Max: Happily, happily we'll include them. One of the distinctions I want to make here, like you were describing earlier, TLA+ is not a programming language. It's a specification writing language. You were mentioning writing down your specifications.

What does that process look like? How do you know that you're writing a comprehensive set of specifications?

Hillel: Would you like some historical background to this or would you like to focus on that one question, because I can do both here.

Max: Let's do the historical background first.

Hillel: Formal verification was originally started with this idea that what you'd be doing is stepwise refinement or correctness by construction. What that means is that you write some sort of specification of your code that describes the system. From that you create a translation of the code. This turns out to be really, really hard. What we're seeing these days is the rise of what's called the flyweight methods or lightweight formal methods, which is saying, okay, we're going to find some way of describing the specification, the design of the system overarching, and say based on these assumptions and this design, we're going to have these things preserved, over this kind of situation, and then leave it up to the programmer to translate that into LiveCode. TLA+ can be used both ways but it seemed the most success being used is flyweight methods because it's so much simpler and easier to do.

Max: In this case, if you were to use TLA+ to verify your methods before writing software. Writing your software product in this ends up being a two-step process where, like you said, you make the flyweight version of your code in TLA+, verify that it behaves like you want and then, the second step of actually implementing this software is taking the design and transcribing it to a programming language that will actually run on a server somewhere. Is that an accurate representation?

Hillel: Exactly.

Max: Got it.

Hillel: That is correct. Sorry, the code that we wrote, the specification was about 60, 70 lines. It ended up translating into about 1200 lines of Ruby.

Max: That is a pretty representative kind of ratio between lines of code in TLA+ spec and lines of code in maybe a dynamic language you'd implement it in.

Hillel: I'd say so. I don't think I have a great statistical understanding of looking across multiple samples but in my personal experience, it's usually a ten to one or twenty to one ratio.

Max: In the Amazon paper, I remember them mentioning the number of lines of code for some of their specs they've

written for S3, DynamoDB et cetera. They don't mention how many lines of code implementation ended up being because lines of code is a pretty hard metric to grasp meaning from, of course. Ballpark, I remember them all being less than a thousand lines of code of TLA+. For our audience that wants a ballpark, these specifications aren't longer than maybe your high school essay.

Hillel: Yeah, pretty much.

Max: You've wrote these specifications at eSpark. You ran them through the model checker. Was this something that you did once up front when you guys were trying to come up with your solution? Or did you run the model checker over a long period of development on the project you were working on?

Hillel: Sort of both. The way that we did this was we originally designed the model of the code and ran the model checker. Then we started writing the code and, of course, when we were writing the codes, we're writing tests, we're writing checks. We're not relying on just the fact that this works in theory to mean it actually works in practice. Then as we start to notice interesting problems, like there's one of those [inaudible 00:18:07] system we didn't consider or they make an adjustment to what they say can trigger the bug. We'd go back and we'd alter the model to make sure this wasn't going to lead us into a pitfall or something. There were a couple of times where we realized that the system that we were building, based on a couple of other assumptions that people were adding afterwards could lead to issues and we were able to redirect the architecture to avoid that.

Max: At the level of the abstraction for TLA+, there's this crossover from writing the specifications to writing the implementation in programming language. How does this jive with what people might understand as TDD, or extreme programming and writing failing tests first? You were describing earlier software verification methodology, where does the use of TLA+ fall in software verification methodology?

Hillel: This is actually a really interesting question and I'm going to try to avoid rambling on it for too long because I

always ramble on this for too long.

Max: Not a problem.

Hillel: The way I'd put it, I think I actually wrote a blog post about TDD and about what we've seen empirically and what we've seen based on the research of how it works and how it's effective. One of the points there that I'm going to paraphrase here is that, they're orthogonal. TDD is a way of writing codes, that way you end up with a huge test suite. TLA+ is a way of designing systems so that way they probably don't have bugs, in theory.

The way that I tend to work is I'll write the TLA+ spec and then I'll basically diagram out the system and then I'll TDD it.

Max: It would be great—and we'll include these in the show notes—links to example TLA+ specs. Perhaps any that you've written. To remind our audience, Hillel made a site, learntla.com, which has examples. I think people will be very interested in seeing examples of what these specs look like.

I think one of the things to address real quick should be how readable are these specs? Are they human readable? I know there's an intermediate language that you can optionally use to write these specs that can be converted to a lower-level specification syntax. How readable are TLA specs?

Hillel: I want to say that they're pretty readable but I think I'm biased here because I've just spent so much time in that world. I think they're readable and the reason I believe that is because I've shown them to other people and walked people through them and they mostly have an understanding of the high level architecture from the TLA+ spec. I think it's not documentation. Nothing's documentation but documentation, but it's a good documentation aid.

Max: One of the ascribed benefits of TDD or PDD test-driven development or behavior-driven development is that your tests are your spec and that, through writing tests you're documenting your code. Outside of maybe yourself and one other team member, would anybody else come

and look at your TLA specs? Or are they pretty domain specific to the software developer?

Hillel: They're admittedly pretty domain-specific and we had a fairly small team and I was working on a small team in a small company. There was some crossover, like I'd share it with project managers and I'd share it with a couple of engineers and we ended up putting them all on repo for longterm preservation. That is really something I do want to explore more, the idea of how it does actually function as documentation. I'm going to say, I hope it does but I can't guarantee that.

Max: Got it. Fair enough. One of the topics that I know you're interested in, and I am too, is some of the more controversial statements that people in the test-driven development community make about how vociferously correct they are. Specifically, there's this one gentleman named Uncle Bob, Robert Martin, who's also from Chicago, Illinois who's made some very religious statements about TDD. Do you mind sharing for our audience about some of the particular virtually religious statements that people advocate about TDD as an approach to software development? What are some of the trade-offs?

Hillel: I'd say the most religious statements I hear are that if you're not doing TDD, you're not being professional. TDD is just like double booking accounting and you'd be thrown in jail if you didn't do double ledger accounting. I probably messed up that name twice in a row. That TDD is actually documentation so TDD, you don't need documentation. Or that TDD acts as [inaudible 00:23:13] proof of correctness. Or that basically this idea that either if you're not doing TDD, you're not writing code correctly, or that TDD is all you need to write code correctly, which are two sides of the same coin here. Go on.

Max: I was just going to ask where do those statements break down because one could imagine that, for example, the double book accounting example can be persuasive for someone who might not think about it very hard because in the land of software consulting, for example, how can you prove to a client that you've done your work, for example? As I understand it, the historical example of TDD was creating a paper trail, not only to be able to prove that

you've done the work that your client asked you to do, in exchange for payment but also to point out that a client is asking for the spec to be changed. To be able to fight back against change requests. Where does the analogy break down with double book accounting and people being thrown in jail for not using TDD?

Hillel: Actually, that's actually a good question and it's actually one of my pet peeves is the comparison between TDD and double ledger accounting. Which, again, I cannot remember the name of the exact term. Basically there's two major problems with that analogy. The first one is I've actually talked to some accountant friend I have and that if you're a small business, you can be totally fine use single ledger and it's not a big deal. Your big companies should be using double booking, but if you're a small company, it's not a problem.

The other problem is that double ledger accounting is the idea that as you're doing your accounting you're verifying it by running a parallel checks on the accounting, right? That doesn't accurately match up to TDD. It's closer to a "test for correctness" technique, which is also very interesting and also has a deep, rich history called contracts, which is mostly out of fashion these days but I think is a very powerful technique and really a lot of people would benefit from it. The analogy doesn't work with TDD and the claim of the analogy that you must use double ledger is itself not quite correct in reality.

Max: Got it. For our audience that doesn't know about contract style testing, do you mind giving people a 20,000 foot overview?

Hillel: Sure. The idea is, you know how with code you might have certain things that you check them and if they fail, you raise an exception then you handle that exception.

Max: Yeah.

Hillel: Very, very high level overview without getting into a lot of the details, like the background. Contracts are essentially that except that you never catch the exception. If the contract is so deep in the code that if it fails it indicates

that there must be a bug in your program, so you stop the program and find it. Turns out the contracts are—

Max: Are these contracts intended to be shipped with production code so that if your production code encounters a bug, they're intended to fail without being caught?

Hillel: Depends on who you talk to. There's a lot of people who think that you turn them off in production because they slow down the code a tiny bit. For that exact reason other reason say that no, if you catch a bug, your program's going to do some undefined behavior, which may or may not be good. You should probably stop that and get a trace that can be sent immediately to the developer so they can write a patch or fix it.

It probably really depends on your problem domain and how much you reliance you need and the assurance you need. It really does depend, again, on your exact use case, I think.

Max: One of the variations that I think our audience may have heard of, and we really have a different name, but refer to the same practice as assertion-based programming. Where you, perhaps at the beginning of a function invocation, make assertions about what your functions' input has received. Is that analogous at all?

Hillel: That's actually the same thing, yeah. You make the assertions and if the assertions fail, you just crash the program. That's how it's-

Max: If you're to write a add function that took two arguments x and y, and you're using non-type safe language like Python or Ruby or JavaScript and you're given an integer and a string, and you had an assertion about the inputs being integers or floats, it would fail? I guess then that would be an example, having an assert statement would be an example of contract-based programming?

Hillel: Yeah. Actually one of the interesting things is that a lot of the ways that assertions are used these days with dynamic languages is as a primitive type checker. That does help because, again, dynamic languages can often benefit from checking your arguments, but you can also do

a lot more with contracts. For example, one of the most typical examples is saying, you can only pass in non-negative integers for this function that withdraws money. Or that this operation on a stack data structure cannot be called if the stack is empty.

Max: For our audience that are curious about ... I think our audience has a gist of how to use contract-based programming or restriction-based programming to add a certain level of software verification. Whether their software will work or not ... For people who are curious about TLA+, like we've been talking about earlier and may go to learntla.com, could you perhaps give people a sense of whether, up front, TLA+, will be valuable to them or not? I think the number of people who work on S3 and DynamoDB, for example, on planet Earth, is not a very large number, but there are plenty of other people who work at companies like eSpark, where you worked or Maidfire, where I work that may benefit from TLA+. What would lead somebody to benefit from using TLA+?

Hillel: That's a good question. I think it's a little broader. I'm going to actually broaden the question a little bit more and say would people benefit from flyweight methods in general? TLA+ isn't the only one. In fact, when people are just starting out, I sometimes recommend they try out a different one called Alloy to start out because it has a lot of power but is also a lot simpler to learn. In general, at least with TLA+, it tends to really shine if your system is one that either changes over time in complicated ways or involves any amount of currency where you have some amount of non-determinism in the program. Those are both the cases where it's often the best tool even in formal methods for doing what you need to do.

Max: Distributed systems is a great use case? I know Leslie Lamport, we should also mention, besides being author of TLA+ and LaTeX is also the inventor of the paxos consensus algorithm. He's very much interested in distributed systems and part of the reason why I would guess that these folks at Amazon adopted TLA+ over Alloy is Leslie Lamport's positive reputation in the distributed systems community. I guess is distributed systems one good use case? You were describing concurrency and one

phrase we haven't used yet, fault tolerance. Is that when people are in the fault-intolerant type of environment they should be checking out TLA+?

Hillel: I don't think that's the only place TLA+ is really valuable but it's definitely one of the most well known and one of the most useful places.

Max: I know in the Amazon paper in the ACM that they mentioned some of the alternatives to TLA+ that they considered for specing out their distributed system problems with S3 and DynamoDB. They mentioned some of the pros and cons and how they ultimately came to choosing TLA+ over Alloy, for example. One of the things they referred to that I didn't really grasp was the expressiveness of the language and how TLA+ was able to express certain aspects of their specification that Alloy was not. I don't know, are there other specification flyweight languages besides TLA+ and Alloy that might exist out there that you're familiar with?

Hillel: Yeah. Definitely. In fact, the case study that Amazon heard about TLA+ is the one that's most public and most well known. They also did another one, which I think is behind a paywall but you can find preprints where they compare it to all the other methods they were looking at while they decided on it. For example, one of the reasons they said Alloy was not the one they chose was because while it was really good for the problems that it is really good for, which I guess is half psychological, the very specific problems they were working on didn't quite fit its model. In general, for example, Alloy is really good if you have to deal with a lot of relationships between data and relational constraints where the time evolution of the system isn't that complicated. They say it's pretty okay if you only have six or seven steps of time, which is honestly pretty good for a lot of cases, but some of the problems they found with TLA+ ended up requiring 35 steps. Other flyweights, the other one in that method, is called Promela or Spin, which was probably one of the first ones to come out but it's fairly limited in what it can do. Another one that came out more recently was called Runway, which was created by Salesforce and the guy who created the Raft Consensus Protocol. I think that people most do [inaudible

00:33:33] self-development on that one. Then there's a few others. There's some stuff that people have been doing with UML actually as flyweight specification. That tends to not be-

Max: For our audience that doesn't know UML, I think we should take a moment to clarify what UML is because we often will use acronyms on the show and never spell out what an acronym is.

Hillel: Yeah, definitely. That's definitely one of the things that I tend to struggle with is going into this massive jargon hole. You know those class diagrams that people draw for objects and those [inaudible 00:34:05] relationships with all the boxes and lines, that's basically UML? It's a lot more complicated and it has its own weird and wild history of why it happened, why it got really popular, why everybody turned against it. The very big overview is that you're drawing lines and boxes and checking copies based on that.

Max: Got it. For people who might be feeling in the stone age and using pen and paper to design the software they're setting out to write.

Hillel: That is great. Pen and paper, honestly most people don't even do that. There's been a lot of work and a lot of studies where either you sit down and write out what you're planning doing with pen and paper, you're going to end up in a much better place than if you don't even do that. If you're doing pen and paper, that's awesome. You're ahead of 99% of developers in the world and you should own that. It's amazing what you're doing.

Max: That's something that in all of the variations of xDD or whatever driven development I've come across. There's readme driven development, which was advocated by one of the co-founders of GitHub. Why they feature readmes so prominently on GitHub repo pages is that if you don't have a clear readme, you're effed. Why would anyone help contribute to you if you aren't putting in a bare minimum amount of time to try and communicate to other human being the value of what it is you're setting out to write software-wise.

Hillel: That plus-

Max: Go for it.

Hillel: Sorry. That plus if you can't describe what you're doing, can you really say you understand it?

Max: Correct. I agree. One of the things that I've been getting more into of late as a result of working at a comic book company is trying to do more visualized representations of software I'm working on. I got a hold of an iPad and started doing digital drawing. One of the things that I'm most impressed by is websites that discuss programming content like Reddit or Acronis tend to have a heavy ... Acronis is less so but Reddit more so. People generally favor visual content. People are way more engaged with visual content and when it comes to educational devices or communication devices, visual representations of what it is that you're interested in are more inherently engaging or viral or entertaining. I think we should encourage all of our audience to, one, grab pen and paper and try and spec out what it is you're writing or trying to develop, on paper first, before even picking up TLA+.

Hillel: Actually have you ever played with Graphviz and Mermaid.

Max: I have not. I've heard of both. Graphviz comes embedded in a number of tools. For our audience that doesn't know, what are Graphviz and Mermaid?

Hillel: Graphviz is a tool that AT&T developed to generally draw essentially diagrams that are called directed graphs where you have certain things connecting to other things and they're different shapes and colors. I've used it a lot for documentation and diagramming and it's really, really nice for that. Again, we're going to include the link. While Mermaid is basically designed for doing flowcharts and simple relations, but it's also a lot simpler. It can be embedded and marked down really easily. They both have their use cases and they're both pretty good tools to mess around with. Again, we'll include links to both of those.

Max: As I recall, they're both command line tools that you point them at your text representation of your graph or chart. It generates a jpeg or png image. I like the image of

markdown-based visualizations where you just write down your visualization, your design flow and markdown file and you get a rich image [inaudible 00:38:13]. I know that in older man pages and older style stuff for documentation you'll often see control flow diagrams that people have typed out in a text editor to try and visualize what's going on. Seems silly that they're not rich images with color and arrows and that kind of thing. I definitely will include these in the show notes. Out of curiosity, I know you have some upcoming speaking gigs, I want to help plug. You're going to be at PyCon this year in Cleveland speaking about what topic?

Hillel: The topic of my talk at PyCon is called 'Beyond Unittests' and it's about, as we've been talking about, other correctness techniques that can be used to check your code. With a special focus on them being the ones that are really easy to get using and start using. The two I'm going to talk about in particular is, as you mentioned, contracts and this thing called generative or property-based testing where you essentially have the computer find test cases for you. They're both very simple techniques that are very easy to start using. They're both incredibly powerful testing techniques.

Max: I've never heard of generative or property-based testing, do you mind giving a 20,000 foot overview of property-based testing.

Hillel: No worries. If I didn't enjoy talking about them, I wouldn't have name dropped them. The idea behind property-based testing is, the best way to describe it is it's a hybrid of formal methods and verification on the nitty gritty, on the ground, like unit testing. What you do is you say, okay, here is your function or your system. Here are the inputs that I want to have. Here is some property that should happen when I do this. A very simple example is if you have a function that reverses a list. If you run that function twice in a list then you get the same list back, right? Then what happens is you specify in your test saying, okay, I'm going to be passing in any list of any integer. What it'll do is it'll check, okay does it work for the empty list, does it work for a list with a million elements. Does it work for a list that has not a number in it. Does it work for a list that

has 50 of the same element and one other one. It goes through all of those pathological cases and sees if any fail. It ends up being a lot broader than say a unittest, but also a little bit less, what's the word, specific. You use a mix of both of them.

Max: Has this been described ever as an extension of unittesting where instead of looking at ... Unittesting refers to a unit under test, a unit of code under test. For example, in my career so far, seen plenty of cases where unittest have been four loops over a range of inputs, making assertions about the range of those possible inputs. I've also seen various packages that don't call themselves property-based testing but are called, for one example, ddt is a Python-library that you decorate your test function with a list of inputs. The test gets re-ran and unique test names are generated based on the inputs of those that you've decorated that test with. Sounds like this is for maybe very wide ranges and ones that you might have to sample for because they're so broad. It sounds like you do an exhaustive search over range basically. Am I getting that right?

Hillel: It's a mix of an exhaustive search plus pathological cases. For example, if you say it can be integers, it will almost certainly try negative numbers, zero and a really high positive number at the very least. What was the name of the library by the way, I'm not familiar with it.

Max: [ddt](#). I think it stands for data-driven tests. I don't think it's tremendously popular. Where people are maybe writing unittests that have some very simple condition that they're checking about a function they have.

Instead of writing a for-loop inside of your unittest, that might fail on the third iteration of your for-loop, by using ddt and decorating your test function, ddt will, by decorating the test function generate n tests where n is the number of inputs that you're testing over. It'll test all n of those instead of just failing at the first iteration in your for-loop that fails. If that makes sense.

Hillel: I'll check it out. Thank you.

Max: Sure. I don't recall it titling itself as property-based testing but I know because you've written about it and others have, about the Python library called Hypothesis. Which is explicitly property-based testing. Do you mind sharing for our audience a little bit about how one uses Hypothesis and why one might want to use it?

Hillel: [Hypothesis](#) is pretty much a property-based testing library. It's a very good one. It's considered one of the best libraries for that of almost all programming languages. The developer, *David MacIver* actually—this is really exciting—he's generalized the backend. He's currently courting it to Ruby and redesigning the way that you can approach any language you want really easily.

It essentially works like that. You say, okay here's my input and my test is going to preserve this invariant and then it'll run through all the tests and it will do the exact same sampling, looking for pathological cases. I think the best example of that, which again we're going to link, is where [somebody was using it when teaching kids Python](#). He was using it to check their assignments of making a password strength checker. He'd say, for example, just generate possible situations where the bug is that it has less than eight characters but can be any other kind of strength as long as it's negative, will it be caught? It was catching these bugs that he never even thought of, where they've let through situations that, in his unittest, were accidentally tripping a different check and not checking the one that they actually wanted to.

That wasn't a really good explanation, I'm just going with the article.

Max: I think it was a good explanation and I think it was a good example. We'll link the article, of course. I think people will benefit from reading it.

Hillel: So many links—there's going to be so many links. You're going to have an entire semester of just links.

Max: Shit, that's better some college courses give you :)

I want to give you a chance to mention that you are currently working on a book for Apress about TLA. Do you

mind sharing for our audience about what it is that you've been commissioned to write?

Hillel: Yes, gladly. I'm writing a book called Practical TLA+. The idea behind the book is that it's drawing from my experiences with Learntla. The thing that I've noticed in a lot of the people who've been learning TLA+ is that they can get the basic ideas down, but they have no idea how to use it to write a specification. Part of the problem there is that there's not that many examples out there of how to write specifications, in part because most of work done is either on abstract algorithms or classified work for companies and governments. What this book is going to be is it's going to be an introduction to TLA+ and cover the language. Then follow it with a lot of examples and several very big case studies that are heavily developed. Like 20 page plus examples that show you to go from ground zero of an idea to a full 300 line specification. One of the actually funny things was I thought that this would be a lot easier. I figured, oh yeah, I already wrote Learntla, I can use that as a launching point, but I've discovered that now that I've had a year more of both using the language myself and teaching and [inaudible 00:46:59] writing. I'm like wow, I'm embarrassed about a lot of the stuff on learntla. Don't get me wrong it's a really good resource but I'm pretty sure I can do a lot better this time around.

Max: I think your audience would look forward to it. I think they should all give learntla.com first, a shot. I'm sure there's somewhere that they can sign up on a mailing list to hear from Hillel when the book is actually published. I'm looking forward to it.

Hillel: If you're interested, and we'll include this, just send me an email. I like getting email, I like talking with people, so yeah, just send me an email saying you're interested. I'll add you to a list.

Max: Absolutely. Well, Hillel, thanks for joining us!

Hillel: Thank you so much for having me. It was a pleasure.

Get exclusive updates in the Accidental Engineer Newsletter:


Subscribe

Categories:


[Careers Testing](#)

The Accidental Engineer

The Accidental Engineer

 [Accidental Engineer](#)

 [AccdtlEngineer](#)

 [Accidental Engineer](#)

Career advice for those starting their software engineering careers. We conduct interviews of experts who have *been there*, making the same career decisions that you are!