Oral History of Leslie Lamport, Part 2

Interviewed by:
Roy Levin

Recorded November 11, 2016
Mountain View, CA

CHM Reference number: X7884.2017

Levin: My name is Roy Levin, today is November 11th, 2016, and I'm at the Computer History Museum in Mountain View, California, where I will be interviewing Leslie Lamport for the ACM's Turing Award winners project. This is a continuation of an interview that we began on August 12th of this year. Morning, Leslie.

Lamport: Good morning.

Levin: What I'd like to do is pick up the theme that we began in the last interview. We were talking about the different threads of your work, and how they wove together in time, and the one that we pursued in some depth was distributed computing and concurrency. What I'd like to do now is to move onto something-- to a topic that definitely wove in with that, which is specification and verification. Which you worked on, it seems to me, pretty much all the way through --I think probably because your mathematical background meant that from the outset, you wanted to prove algorithms correct, not just create them.

Lamport: Well.. I guess last time I didn't get to the story about the bakery algorithm that I..

Levin: I'm not sure, but why don't you give it to us now?

Lamport: Well.. when I first learned about the mutual exclusion problem, I think it may have been when.. in 1972, when I believe I joined the ACM. And in one of the first issues of CACM that I received, there was a paper giving a new solution to the mutual exclusion problem. And I looked at it and I said, "Well that seems awfully complicated, there must be a simpler solution." And I sat down, and in a few minutes I whipped something off, it was a two-process solution. I sent it to the CACM, and a couple of weeks later I got a reply from the editor, who hadn't bothered to send it out for review, saying, "Here's the bug." That taught me a lesson. <laughs> Well it had two effects. First, it made me really mad at myself, and I said, "I'm gonna solve that damn problem." And the result of that was the bakery algorithm. But it also made me realize that concurrent algorithms are not trivial, and that we need to have a really good proof of them. And so that is what got me interested in writing proofs. And I think it made me different from most computer scientists who have been working in the field of verification, in that my driving influence has been, that I wanted to make sure that the algorithms that I wrote were correct. Now you think everyone who was any good in the field of concurrency, starting from Dijkstra, who was certainly very good in it, understood the need for writing proofs. And so they were writing their own proofs. But.. Dijkstra-- well other computer scientists -- and I think even Dijkstra, did not consider -- let me start that again. Other computer scientists working on concurrent algorithms, did not approach the task of a formal method for proving correctness of concurrent algorithms. Dijkstra was interested in correctness of programs, but the work of his that I remember was only on sequential algorithm, that is, the formal method. And the other early people in the game -- Ed Ashcroft, Owicki and Gries -- were interested in formal proofs of concurrent

algorithms, but weren't writing concurrent algorithms themselves. So I think I was, from the start, rather unique in having a foot in both fields.

Levin: And would it be fair to say that over the ensuing decades, your approach to the formalization and specification of algorithms evolved quite considerably -- probably as a result of experience, but perhaps due to other factors as well?

Lamport: My view of my.. progression of ideas, was that of a.. basically a slow process of getting rid of the corrupting influence of programing languages. <laughs> and getting back to my roots, which was mathematics.

Levin: Yeah, I want to talk about that a little bit more, because I think that's one of the interesting things, and personally I've heard you talk about this quite a bit over the years: the fact that programming languages tend to get in the way of understanding algorithms and certainly, as you just said, in terms of proving them correct. Maybe if we start, as you mentioned, Ed Ashcroft and the Owicki-Gries method. Maybe if we start by talking a little bit about that, and the approach that they took: how it influenced you, how it contrasted with what you ended up doing.

Lamport: Well, Ashcroft took the very simple approach of having a single global invariant, which I eventually came to realize was the right way to do things. But.. Susan Owicki and David Gries and I, were influenced by-- well I was influenced by Floyd's paper *Assigning Meanings to Programs*. And they were I think influenced by that, and by Hoare's work, on Hoare logic. And we both came up with the same basic idea of.. as in the Ashcroft method-- I'm sorry, as in the Floyd method, attaching assertions to control points in the program. The one difference is that I realized immediately that the assertions needed to talk, not just about the values and variables, but also the control state. And so right from the beginning, that was encoded in the algorithms. Owicki and Gries were under the influence of Hoare, and I think the whole programming language community that-- Well perhaps it's time for a little digression into what I call the "Whorfian syndrome." The Whorfian hypothesis..

Levin: I think you did talk about that last time, actually.

Lamport: Oh, I did? Oh.

Levin: I think so, yes.

Lamport: Oh, fine. Then.. I would say that Owicki and Gries suffered from the Whorfian syndrome. And one symptom of that, is that if the programming language doesn't give you a way of talking about something, it doesn't really exist. And since the programming languages didn't give you any way of talking

about the control state, then it didn't exist, so you couldn't use it. So instead they had to introduce auxiliary variables to capture the control state. But what they did is.. they were really doing-- they and I, were really doing a generalization of Floyd's method. But they pretended, and I think perhaps even believed <laughs>, that they were actually generalizing Hoare's method. And when they were doing.. Floyd's method in Hoare clothing, things got really bizarre. I mean how bizarre it is, is that they talk about a program violating a proof of another process. The very language. And if you just stop back and think <laughs> of this idea, you know, a program violating a proof. "I'm sorry, but you can't run your payroll program on this system, because it violates our proof of the four color theorem." I mean..

<laughter>

Lamport: But that's what they talked about. And as a result, things got awfully confusing. And even Dijkstra was not immune to that, he wrote an EWD calling something like, "A personal view of the Owicki-Gries method." In which he was explaining it. And he, I think, to the very end <laughs> was proud of that paper. And I looked at it and said, "My God <laughs>. How could people possibly understand what was going on if, you know, reading it from that?" Because if you explain it in terms of the annotation of the program, being a representation of an invariant. And then the basic invariance-- the basic way you reason about an invariant, is you show that each step of the program preserves the invariant. And when expressed that way, and it was very obvious when-- in my approach, where the control state was explicit. And in fact, in my original paper it explained what the global invariant was, everything is really simple. But I think there was a period of about 10 years when people just really didn't understand what the <laughs> Owicki-Gries method was all about. And most people were unaware of my paper. And that seems to have been sort of-- I think it was about 10 years afterwards that people started citing my paper, and presumably that meant they might have read it <laughs>. And then I think it became clear to people what was going on. And by that time, I think I had pretty much abandoned the idea of scattering the invariant around, you know, by putting it in pieces of the program. And just instead writing a global invariant, just like Ed Ashcroft <laughs> had done before us.

Levin: Just to clarify, for my own thinking. At the time that all of this was going on, which I think was probably in the '70s, or at least the late '70s, the focus was chiefly on sequential programs? Is that right? That most of these methods were not really trying to deal with concurrent programs?

Lamport: Oh no, Owicki-Gries was dealing with concurrent programs..

Levin: Okay.

Lamport: …that was their whole thing.

Levin: Okay.

Lamport: And extending.. Hoare to concurrent programs.

Levin: Okay. But Floyd's method was originally intended as a sequential programming..

Lamport: It was originally just for-- Floyd and Hoare..

Levin: And Hoare, yes.

Lamport: …originally intended for.. concurrent-- for sequential programs. Actually.. I came up with an actual generalization of Hoare's method for concurrent algorithms. I think I did it, that is I think the original paper was by me. And Fred Schneider and I wrote one or two papers, about what I call the "generalized Hoare logic." But I believe, and I should check the <laughs> record, that the original paper was mine. And when I sent that paper to Tony Hoare-- and I wish I had saved the letter, in those days people communicated by letter.. that Tony replied. And he essentially said, "When I did the original Hoare logic, I figured that its extension to concurrency would look something like what you did. And that's why I never did it."

<laughter>

Lamport: And at the time of course, I thought, "Oh, an old fart <laughs>, you know, what does he know?" But in retrospect, now I agree completely.

<laughter>

Lamport: It's just because.. I think the global invariant approach is the best way to do it.

Levin: And the global invariant approach doesn't particularly single out sequentiality versus concurrency, it's just an invariant.

Lamport: Right.

Levin: So it spans those two areas, which..

Lamport: Well.. I don't think people thought about the Floyd approach in terms of a global invariant.

Levin: Mm-hmm.

Lamport: Because people were thinking, you know, one thing at a time, so you're going from here to there, from one assertion to another. And I'm not sure, but I suspect that Ed Ashcroft was the one who understood, who realized that the Floyd method was a single global invariant.

Levin: Mm-hmm. Now you ended up working with Susan Owicki quite a bit subsequently. Did that grow out of your.. I won't say "conflicting," but somewhat differing views about how to do these things? At least back in the '70s?

Lamport: Oh. Well what happened is that this was all going on around.. actually I believe it was '77 that I published my paper, I think. David and Susan's was published in '76. Just.. they got it published a little faster <laughs>. And in '77 Amir Pnueli published his paper on temporal logic, introducing temporal logic to computer science. And Susan was teaching at Stanford then, and she decided to hold a seminar on temporal logic. And my initial reaction to temporal logic was, "Oh.." it was just this formal nonsense. But I said, "What the hell? This might be interesting." And so I.. attended the workshop, or what is it called? I guess a seminar. And what I came to realize, and Susan came to realize as well, was that temporal logic was the right way to reason about liveness. So for the TV audience, a little digression. Correctness properties-- by a correctness property, the kind of properties that I've studied, and that people usually mean when they talk about correctness, are basically assertions that are true or false of a single execution of the program. That is, you can tell whether this property is satisfied, by just looking at a single execution, as whether it's meaningful to talk about it being satisfied by this execution. For example, it never produces the wrong answer. Well if it's wrong, there's a behavior, an execution that says, "It produced the wrong answer." Or it never produces an answer. Well you have to look at an infinite behavior, but you can look at that one behavior and say, "It didn't produce the answer, so it didn't satisfy the property." There are other properties, like average case behavior for example, that are not correctness properties in that sense. So when you talk about correctness property, that's the property that I mean. And it turns out that every property can be expressed as the conjunction of two different kinds of properties. A safety property, which intuitively says that something bad doesn't happen, it doesn't produce a bad answer. And a liveness property, which intuitively says that something good eventually happens, like it terminates. Now in my original paper, in addit-- the Owicki-Gries method deals only with safety. My original paper dealt with-- considered safety properties, by prov-- as invariance properties. Basically, essentially the same way as the Owicki-Gries method did. But I also had some method for proving liveness properties. And I'm not sure at the time how happy I was with that method. But certainly in retrospect, in looking at.. once temporal logic was there, once Amir had shown how to use temporal logic, it was clear that temporal logic was the way to reason about liveness properties. Because it provided a way to use safety properties, combined in proving liveness properties. Which is something that you have to do. And Susan and I.. I guess we were talking about that at the time. And so we published a paper on using temporal logic to.. prove liveness properties. The other work on temporal logic that I did, was a paper called *"Sometimes" is sometimes "not never."* <laughs> It makes sense when you put some quotation marks around a few of the words.

<laughter>

Lamport: Because I realized that there was a confusion going on. Because there are actually two different styles of temporal logic, which are called “branching time” and “linear time.” And computer scientists tend to naturally think in terms of branching time, whereas ordinary language tends to be based on linear time. The difference is that if you say something is not al-- what does it mean to say something is not always false? In branching time logic, it means that it’s possible for it to be true. And in linear time logic, it means that it must eventually be true. And I realized that, well, the logic that Amir had introduced was linear time, and I realized that that was the right logic for talking about dealing with safety. And I wrote a paper with, you know, it had a couple of not terribly deep or interesting theorems to make it publishable. But the basic point of the paper, was to point out those two different kinds of logic. And to illustrate why I thought, that linear time logic was the right one to be using, for proving correctness properties. Pause, while I think if there was something that I meant to say.. no, I lost it.

Levin: So you mentioned that Amir used linear time, rather than branching time..

Lamport: Oh yeah, there..

Levin: Yeah sorry, go ahead.

Lamport: Yeah, so there’s an amusing anecdote about that that was told to me by Dominique Méry, who was I believe a student of Patrick Cousot’s.

Levin: I’m sorry, say that name again, I didn’t get it.

Lamport: Patrick Cousot.

Levin: No, before that.

Lamport: Dominique Méry.

Levin: Oh, yes.

Lamport: Who I believe was a student of Patrick Cousot. And when he read my paper, he said, “It was all wrong. It doesn’t make any sense.” And the reason was that he was translating “eventually” into the

French word.. “éventualité” which-- “éventuellement.” “Éventuellement” in French means “possibly.” <laughs>

Levin: Wow.

Lamport: And it somewhat does in English too, you can talk about an eventuality being a possibility. So <laughs> he was misreading what I was saying, and it made no sense to him.

Levin: Aha, aha.

Lamport: And I’m not sure how Patrick became, you know, realized his mistake.

<laughter>

Levin: Interesting. But when Amir chose to use linear time rather than branching time, do you think that was a conscious choice because he understood, as you came to as well, the appropriateness of that for liveness? Or was this is some sense because it seemed like it was the intuitive notion of ordinary language?

Lamport: Well my understanding, and I’ve never confirmed this with either Amir or Nissim, but Nissim Francez wrote a thesis under Amir, in which he was proving properties about concurrent programs. But he was doing it in terms of explicitly talking about time. Which meant that all his formulas had a whole bunch of quantifiers, “For all, you know, time T, there exists a time S, greater than T such that for all times U greater than S,” blah, blah, blah. And all these quantifiers, and Amir realized that those particular values of time, were not the interesting thing. And I suppose he must have known a little bit about temporal logic. And he realized that what temporal logic was about, was in some sense putting time into the logic so you didn’t have to make it explicit. And I think it must have been clear that when you translated what Nissim was doing into temporal logic, you got linear time temporal logic. Now I don’t know how-- whether Amir was aware that he was making a conscious choice of the particular kind of temporal logic. I suspect he was, because the logic he’s using has a name in the literature of temporal logic, you know, something like, you know, something four point five point seven in somebody’s book <laughs>. And I think that temporal logicians realized that that was a logic of linear time. But.. however he got there, it was clear that that was the way to translate what Nissim had been doing.

Levin: Mm-hmm. So after you-- and working with Susan -- sort of got this idea of applying temporal logic as the machinery for liveness properties. Where did that lead next?

Lamport: It had no immediate consequence. In the sense that I think it made pretty clear how you would use temporal logic, in the context of doing the kinds of reasoning about programs that the Owicki-Gries method was doing for safety. And how to combine those safety properties.. how to use the safety properties, the invariance properties, improving liveness properties. Oh, and I shouldn't say that it had no-- well I think the work of Susan and me had no immediate consequence. But Amir's work, of course, had enormous consequences. One of the initial ones that it had was-- and I think the paper by-- probably the next, I would say important paper on the use of temporal logic in reasoning about programs, or at least talking about programs, was the paper by Richard Schwartz and Michael Melliar-Smith --I don't remember the name -- in which they advance the idea of describing the entire program with a temporal logic formula. And I was-- that idea sounded great. You know, wouldn't it be wonderful, to be able to just represent the entire program as a single formula that you could reason about? But while it sounded great in principle, it just didn't work in practice. And what I saw was.. Richard and Michael, and Fritz Vogt, who was working with us at SRI for the year, I believe for a year. And they spent about two days trying to write a temporal logic specification of a FIFO queue, a first in, first out. I mean the world's simplest example of a concurrent system, you know, two-process system. And they weren't able to do it. They were able to do it, if they made the assumption that the same element was never put into the queue twice <laughs>. And in fact, I think Steve.. German-- I don't remember if his name Jerman or Gerrman <laughs>. He proved a number of years later that in the temporal logic which they were using, which was the original temporal logic that Amir introduced, it was impossible. And so they started-- people started inventing new temporal operators. An "until" operator and-- used to have a slide <laughs> with all the temporal operators and including, someone said, "Every other Tuesday." <laughs> And I just realized that that was not going to work. That basically, specifying-- trying to specify-- I'm not sure I was aware that it was safety problems <laughs> that were the issue. But trying to specify safety with temporal logic formulas, is a loser, it doesn't work. Because what you wind up doing is writing a whole bunch of axioms. And even if the individual axioms are comprehensible, what you get by combining a whole bunch of axioms is totally incomprehensible. And if you want to see an example of that, look at some modern specifications of weak memory models. And some of them --I know the Alpha model was one, and the Itanium model was another -- where they basically specified the possible outcomes, from a series of read and write operations done by multiple processes, using a bunch of axioms. And I've seen people puzzle for days, over whether some particular sequence of two processes each executing three instructions, whether a particular outcome was allowed by those axioms or not <laughs>. And of course, as you remember from our SRC days, Jim Saxe discovered that the original axiom systems were-- the Alpha permitted cyclic time dependencies where one process wrote a certain value, because another process, which read the value it wrote, did something that allowed it to write that <laughs> process. But I realized that the only way to describe the safety properties of nontrivial systems, precisely and formally, was basically using some kind of state machines. And that's what programs are. If you look at the semantics of programming language, you know, what's called the "operational semantics." -- which is the only practical ones that people I think do these days, when they want a real semantics for a real programming language -- they are describing how to interpret a program as a state machine.

Levin: So your approach essentially evolved to separate the specification of the safety properties from the specification of the liveness properties.

Lamport: Exactly.

Levin: The latter being with using temporal language, the former being state machines and conventional logic.

Lamport: Yes.

Levin: First order logic.

Lamport: I thought things were, you know, worked really well. Until sometime, it was in the late '80s, I started to write a book on concurrency. And I know I was able to stand at the whiteboard, and show you how to prove something <laughs>. And totally convincing, you know, how you do the reasoning about the safety properties of using a state machine, and then use temporal logic liveness properties. But when I started writing something down, well when you write something down that really forces you to do it precisely. And when I started doing it precisely <laughs>, I realized it didn't work. It just didn't hold together. And I don't remember exactly where things were breaking down at this point. But at any rate, what that eventually led me to do, was to invent TLA, which is a temporal logic. And TLA starts with-- it's a generalization of Amir's original temporal logic. Except everybody else tried to generalize it by using more complicated temporal operators. What I did, is that Amir's and all other temporal logic that I know of, the fundamental atomic building block that you use for temporal formulas, were assertions about a state. So a temporal formula is an assertion about a sequence of states. And you built them up out of fundamental building blocks, which are assertions about a single state. What I did in TLA, is generalize from an assertion about a single state, to an assertion about pairs of states: a state and the next state. And that allowed me basically to write a state machine, as a temporal logic formula. And so I could then describe the entire program as a single temporal logic formula, except in a way that really worked in practice.

Levin: So we should probably say that the A stands for "actions," which are these states pairs.

Lamport: Yes.

Levin: …that became the basis for your logic, for this kind of reasoning.

Lamport: Yes, it's the Temporal Logic of Actions. Some people thing it stands for "three-letter acronym."

<laughter>

Levin: You probably weren't thinking about that at the time..

Lamport: No, I wasn't.

Levin: …but maybe you were <laughs>. So we're now in the early '90s, is that right?

Lamport: Yes.

Levin: And about that time, you did some other work that's in the specification area and maybe not directly related. But I want to ask you about that, because it was a notion that became quite important. And that was refinement mappings: work that you did, I think, with Martin Abadi.

Lamport: Yeah. Well refinement mappings started out being-- well they were originally done semantically -- in which the program-- I should probably stop calling them programs and just call them systems, because they're more general than programs. And my interest-- well what I realized, is that the things that I had been calling programs, and that other people had been calling programs back in the '70s and stuff, they weren't programs, they were algorithms. They were not things that you could throw into a compiler and run. And I realized at some point that, back in the '70s when we started, what we were doing were proving correctness properties of the program, in terms of the program itself. But I realized, and other people realized as well, that we should really be describing what the program should do, independently of the program and then prove that the program satisfied the specification. Now that was one of the wonderful promises of temporal logic, is that if you did that, and if the program could be represented by a temporal formula, and the specification could be represented by a temporal formula, then ideally you could, proving that the specification-- that the program implements the specification, really means proving that the formula that described the program, implies the formula that describes the specification. And so you've reduced it to a simple mathematical relation: implication. But that didn't work when, you know, you couldn't describe practice programs or algorithms, in terms of the temporal logic that was being used at that time. And so that idea was sort of.. was dropped. And when Martin and I were working on refinement mapping, we were looking at the problem of proving that an algorithm or system implements its specification. When they were bo-- when the safety parts were represented as state machines. And when it-- when TLA came along, it was trivial to turn this semantic approach, in terms of state machines, into a formal logical approach. And all of the reasoning that we were doing could be done completely inside of TLA.

Levin: Mm-hmm. So this idea of refinement mapping has had quite some durability.

Lamport: Yeah.

Levin: I think.. it was early '90s I think, when you published the paper, the first one with Martin.

Lamport: Yeah. I should explain that the idea of refinement mappings is a generalization of data refinement, which was introduced by Tony Hoare in the late ‘70s. Maybe around ’78, I think. He called then, I think, “abstraction functions.” And the difference between refinement mapping and an abstraction function, which is in some sense the difference between reasoning about sequential programs and reasoning about concurrent programs. Which is the distinction between the Hoare logic, and reasoning in terms of global invariance, is that instead of the refinement in abstraction functions, you just look at the mapping at the beginning of the execution and at the end of the execution. But with the refinement mapping, you’re doing that abstraction function at each step, and it has to work at each step.

Levin: Mm-hmm, got it. And as you mentioned, this could be done-- these refinement mappings could be done completely within TLA.

Lamport: Yes.

Levin: And so I imagine that’s what you in fact ended up doing, as you continued to use and develop TLA through the ‘90s.

Lamport: Well, what I was doing with TLA in the ‘90s.. I understood that TLA was the correct logic, for describing a program. But I was still suffering from the straightjacket <laughs> of programming languages. I hadn’t escaped from that. And my idea is that-- my original idea was that I would have some specification language, which would use the usual programming language constructs, assignment statements and stuff like that. And then that would be translated, or.. as a TLA formula, which one would then reason about. But actually one significant step came from Jim Horning, who said, “Instead of using assignment statements, use..” I don’t remember what he called them, but what are now call-- I now call them “TLA+ actions.” Rather than an assignment being, you know, you assign a new value to the variable, based on the values of variables in the current state. You just write a relation between old values and new values. So that instead of writing, “X gets X plus one,” you just write the new value of X, which I now write “X prime” equals the old value of X, which I now write as “X” plus one. And I think.. Jim was writing primes and unprimes as well, as in fact I had been, in the 1984 paper. But I had actually forgotten about it.

<laughter>

Lamport: There’s an amusing story.. that when I published TLA and talked about describing semantically actions, as relations between primed and unprimed variables. A computer scientist, whose name I will not try to remember, said I should credit him with that idea. Because it appeared in a paper of his. And it seemed to me that the idea of using prime variables and unprime-- for new values and unprimed variables for old values, you know, must go back to the ‘70s. And so I did as much of a literature search as I was going to do. And the earliest reference I discovered, was a paper of mine..

<laughter>

Lamport: …from I think 1983.

Levin: Maybe Jim Horning read that paper.

<laughter>

Lamport: Anyway, that was a digression. So Jim convinced me to try writing them in terms of-- as primed and unprimed variables. And I figured I would still need a bunch of programming language-type ideas. But I didn't know which ones. So I decided that I would just write them in TLA plus, and when I needed some programming language construct okay, I will use it. One of the things-- the realization that I had at some point, was that I simply assumed that, like any computer scientist does, that you have a programming language, it should have types. And what I realized is that I didn't have to add types into the language. I could instead-- type correctness could be stated as an invariant. And I was wondering, I mean I was so tied to them I just sort of asked Martin, Martin Abadi, "Well I don't need types, because I can just use invariants. But should I use types?" And Martin said, "Well if you can do it without types, that would be more elegant." And so I decided I would do away with types. And boy, was Martin right on that one. Because what I discovered is that if you do things-- try to do things rigorously with types, you really either have to-- I couldn't do it rigorously with types, without enormously restricting the expressiveness. And in fact other people, computer scientists who think they're doing things rigorously with types, are fooling themselves. At about that time, somewhere around the late '80s or the early '90s, there were three books that came out that dealt with concurrency of programs carefully. One of them was the Misra and Chandy book on UNITY. One of the was Apt and Olderog's book on reasoning about concurrent programs. And the other one was the Gries and Schneider book, which is discrete math, but it talked a lot about those. And they all used type systems. And I came up with this very simple question. Basically, one way of thinking about it is, exactly what does the statement "X prime equals X minus one" mean if X is of type natural, and the value of X is zero? So what does "X gets X minus one" mean? And it's an error. That's not a meaning.

Levin: Mm-hmm.

Lamport: That's not, you know, giving something meaning means mathematics and, you know, you simply can't have a mathematics where you have to prove a theorem in order to determine whether something is syntactically legal. That's nonsense. Well neither the Chandy and Misra, nor the Gries and Schneider book, answered that question. Even though they thought they were doing things really rigorously and completely. Apt and Olderog understood the problem. And the way they solved the problem, was not allowing a type "natural number". They can only have a type "integer" <laughs>. But it turns out that's not a problem in an untyped system. So when I got rid of types, then I just realized that

mathematics was all I needed. I didn't need any of this programming language stuff, like types and assignment statements and stuff. And it turned out that there were some things that were useful, that came from programming languages. For example, declaring variables. Because it turns out to be useful, both for parsing for example, and it's a nice way of sanity-checking things. And there are other things, like a way of splitting things into modules. There's a lot of things mathematics wasn't—well, the fundamental problem mathematicians hadn't really addressed, is how do you deal with formulas that may be hundreds of lines long? Because mathematicians don't write them. And a specification can be-- is really a mathematical formula that can be a hundred or a thousand lines long. Which sounds terribly-- if you say that to a programmer, they say, "God, a thousand-line formula! How do you can you possibly understand it?" But if you tell them, "Oh, and a thousand-line C program." "Oh, that's trivial."

Levin: <laughs>

Lamport: Well, C is a hell of a lot more complicated than mathematics. So why is a thousand lines of C program trivial, and a thousand lines of mathematics not? It's because mathematicians hadn't developed-- hadn't really thought about that problem. And in fact, mathematics has the most wonderful method of hierarchical structure—well, you deal with complexity like that as hierarchical structuring. And math has the most wonderful, most powerful hierarchical structuring mechanism I know of. It's called the definition. I mean if you look at a calculus textbook or something, you get to numbers, to the derivative with about three definitions <laughs> or something that are built on top of one another, so the definition is enormously powerful. And in fact mathematicians didn't even have any formal notion of a definition. Like I don't know-- I mean I'm not an expert on logic, I mean in fact I'm an ignoramus about logic. But the logic books I've looked at <laughs>, I've never seen any formal definition of a definition. I think the closest they come, is that they will write definitions as axioms. And I don't like that <laughs>. So I introduced the precise notion of a definition, and things-- and a few other pieces of notation that mathematicians didn't have. For example, mathematicians had no-- talk about functions, but they don't provide a practical way of defining-- of writing a function. For example, the function whose domain is the natural numbers that maps any number X, into X plus one. There's no way of writing-- in ordinary mathematics, no way of writing that function rigorously. Unless you go to the definition of a function as a set of ordered pairs, which is not a very nice way of doing it. But any rate, things like that that I had to add to the language, and the language I came up with is called TLA+.

Levin: Mm-hmm. And you've just been talking about the fact that mathematicians typically don't deal with these hundreds-of-lines formulas. But in the work that you were doing, you of course had to. And that actually led to some approaches in how to write a formula, for example. Which is something you actually published a paper about..

Lamport: Yeah. And also how to write proof. Somewhere along the line, being educated as a mathematician, I was under the illusion that the proofs that mathematicians write are logical. And what I came to realize, is that the mathematical proof is a literary style. That when viewed literally, contains a lot

of nonsense, things that make no sense. But mathematicians learn to read this literary style, and understand the logic behind it. Non-mathematicians don't, and that's a major reason why they can't learn-- why non-mathematicians can't write proofs. But also, I discovered that in practice for writing hand proofs-- the paragraph-style proofs that mathematicians write simply can't handle the complexity that's involved in the kind of proof that you have to write for even a hundred-line-- a specification that's a hundred-line formula, and so I developed a hierarchical proof structure.

Levin: Another way of dealing with the complexity of big things is a kind of divide and conquer, or what we might call modularity or decomposition and so on, where you build up things from smaller pieces and put them together, and I think you did some work with Martin again on how you do that with specifications.

Lamport: Yeah, I think-- well, there was a paper-- well, we wrote two papers about that. I think the second one was in terms of TLA, although that can be done with any way of writing specifications. I haven't been working on that, because basically, the world is far from the state where the kinds of-- where one needs to modularize TLA+ specifications in that way; that is, by writing them as independent entities that get combined, where-- so you can sort of-- how to take this specification of FIFO queue and then have it on the shelf and then when you wanna use the FIFO queue inside something else, you just combine it literally with that specification of whatever it is you're building. What you-- what people do these days and what I advocate is that basically, if you want to use a FIFO queue, you have a specification of a FIFO queue, you basically copy the relevant pieces of that specification into your current specification and it's a very straightforward procedure, but it's cut-and-paste, it's not modular composition. But if your biggest specifications are one or two thousand lines, cut-and-paste works fine. Also, the major tool we have for-- engineers use for checking TLA+ specs is the model checker and that won't handle specs that are written in that modular style.

Levin: So this leads us in the direction that I wanted to get to next, which is the practical use of TLA+ and the tools that one needs in order to make it useful for at least some people, and that-- I think you've just touched on that a little bit -- and the pragmatics of that, in particular needing to have a model checker. The model checker came along, actually, sometime earlier and you worked with several different people over time on that, right?

Lamport: No, no.

Levin: Am I confused?

Lamport: You're confusing the model checker with the proof system. There was a TLP proof system that was of the logic TLA, not for the language TLA+, and it was quite primitive, but it was in some sense a proof of concept that you really could reason about TLA formulas rigorously enough that you can check

them with a mechanical theorem prover, although it was very primitive. What happened for the major tool, the model checker, is that I was approached by some people in the hardware group of-- I don't remember if it was DEC or Compaq -- it was probably Compaq at the time -- but from the old DEC people and they were building a cache coherence algorithm, or they were using a cache coherence algorithm, that was the world's most complicated cache coherence algorithm and they wanted some assurances that it was correct, and they asked me and so I volunteered and got Yuan Yu and Mark Tuttle and someone you don't know, Paul Harter, who was not in our lab, to join me in writing a-- trying to write a correctness proof of their cache coherence algorithm. Well, we slaved on that for six months and all we did was find some-- one really, really tiny bug in their algorithm, and that's because those guys who did that algorithm were incredibly bright, unlike some of the people, the engineers, who came from Compaq Houston who tried to build a cache coherence program-- protocol and they gave a talk about what they were doing at WRL, our sister lab in Palo Alto, and-- with a much simpler algorithm and somebody at the lecture in real time pointed out a bug in their algorithm. But those guys, the East coast from DEC, were really good. But any rate, Yuan was convinced that all of the -- kind of stuff we were doing should be done by a machine, so he decided he was going to write a model checker and I said, "Oh, you couldn't possibly do it. It's not gonna work. It's not gonna be efficient enough." But fortunately, he ignored me and he went ahead and did it and it's the model checker that has made TLA+ of practical interest to engineers.

Levin: So this was TLC. I have-- I don't have a actual date for when that happened, but I think you're right; it must've been in the early days of Compaq ownership of DEC or maybe even a little before that.

Lamport: No, that was around 90-- I think '99, approximately.

Levin: Okay, so right in that period of time, yes.

Lamport: Well, not too long before we left Compaq.

Levin: And so by this time, is it fair to say that the-- that TLA+ had matured to the point where other people were at least trying to use it -- it wasn't just a tool for you?

Lamport: Oh, yeah, the-- in fact, its first use as it was being written was by-- in checking the cache coherence protocol of the EV7, the next generation of Alpha chip, and those six months that we sweated were not in vain, because they gave us cred with the hardware people. They knew that we were willing to put our ass on-- our asses on the line, and so when we had this model checker, the-- a manager of-- (I think he was managing the testing of the EV7) agreed to assign a junior engineer to write a TLA+ spec of their algorithm and check it, and the-- those people from DEC went-- when DEC sold the-- their hardware processor business to Intel, they went over to Intel and they started the use of TLA+ at Intel and it was used for quite a few years there, and I've lost contact. I don't know if it's still being used there or not.

Levin: And you wrote a book about TLA+ around this time too, right?

Lamport: Yeah, the-- right, it's about-- yes, it was after the model checker was written, but I think it was being written while-- it started before the model checker was written, so that the book wasn't based around the model checker the way it probably would've been if I had written it later. The model checker was a chapter in the book, but the book was about TLA+. But I think in retrospect, it's lucky that I didn't appreciate the power and the usefulness of model checking, because I was-- thought that, "Oh, you need to do proofs, because model checking can only check tiny models and real system." I didn't appreciate how effective even really tiny models were at finding bugs in algorithms, but the upshot of that was that I designed TLA+ before the model checker was written, and had I-- and that made it a much better language because it would've been tempting to do things like add types and other things that would make it easier to model-check specs. As it is, it was, in the early days, a handicap for TLA+ because not being written for model checking, it was much slower to model-check than lower level, less expressive languages. That advantage has been way reduced by technology because the cost of the actual computation has become much, much smaller and the overpowering cost is dealing with the states and writing out to disc and stuff like that, so TLA+ is not a big handicap in terms of efficiency of model checking these days.

Levin: And so the model checker was a primary tool, then, in making TLA+ usable by...

Lamport: Yeah.

Levin: ...shall we say, mortals, but that wasn't the end of the story about tools. I think that the system continued to evolve after that, and can you talk about that a little bit?

Lamport: Well, there are two evolutionary steps, first, the toolbox, which is an IDE-- Integrated Development Environment -- basically a GUI for writing TLA+ specs. And I mean, I was told by Gerard Holzmann that when he added the most primitive GUI to his Spin model checker that hardly did anything to you, the number of users increased by an order of magnitude, so I said...

Lamport: ..."Duh, what should I do about that?" At any rate-- and it's made things a lot easier, especially made it much more convenient to use the model checker, and another thing is, we have a prover project going to be able to write TLA+ proofs for-- and mechanically check them. My original interest in that was to give TLA+ academic cred, thinking that it would be-- people would be more likely to use it and teach it in universities if there were a theorem prover attached, and I was rather surprised at how good the theorem prover turned out to be, and I was able to use it to prove really nontrivial-- and verify really nontrivial algorithms, and it became even better when we managed to hook up SMT solvers to it. So now, for proving safety properties, I think it's practical, at least for journal-sized algorithms, and in fact, one journal algorithm was-- that was involved with as sort of the verifier (namely, I wrote and checked a

written form of proof) but actually, a couple of the authors got hooked on it and they, you know, towards the end, they were writing parts of the proof, too. So it's work, but I think it's not something that engineers will be doing-- using for-- I don't think for the kinds of algorithms they write, although they are interested. They would like to because, as effective as model checking is, there are systems that they build that they can't check in a large enough model to give them the kind of confidence that they would like, and they would like to be able to write proofs, but as far as I know, nobody has-- in industry has tried writing a proof.

Levin: You mentioned academic cred. Can you say how that has played out? Has in fact TLA+ and its-- and the system around it come to be used in teaching?

Lamport: Well, I'm afraid it hasn't. The problem is that verification or formal specification is in academic completely separated from the system-building side of education. And the most-- I think most of the courses that teach specifications-- teach specification are actually teaching specification languages, and yet nowadays they'll spend two weeks on TLA+ with as well as all the other <inaudible> that they will do. But I don't think that students come out with any real experience in how specifications can be used in writing real programs and building real systems.

Levin: Little peculiar, perhaps, in that if one is studying systems, one learns about tools for expressing the actual code of those systems and you obviously have to know about programming languages so you can write a system and so on, but in terms of the algorithms part of it, it seems to be neglected.

Lamport: Well, actually, I should take that back. There is a little interest in it, for example, the Raft algorithm. They wrote a TLA+ spec and model-checked it. I think we're not terribly satisfied with it. I think there was one bug that it didn't catch, but I'm not sure if it didn't catch it because of the model they used or because-- that is, because of the spec they wrote or because they couldn't chest it on-- test it on a large enough model, but my suspicion is that learning to write the kind of abstraction that you need in order to be able to simplify a system down to an algorithm -- down to its essentials -- is an art that has to be learned, and I think if you just try to do something like Raft as your first specification, it's not terribly surprising that you're not gonna be able to write a good enough spec to-- or an abstract enough spec to be able to catch the errors you should be able to catch.

Levin: Is that perhaps one of the problems that-- let's say one of the obstacles that-- to use of specification technology, that programmers, engineers aren't sufficiently comfortable with abstraction at the right level?

Lamport: Well, one of the-- one time, I asked Brandon Batson, who was one of the engineers from DEC who went to Intel, who was one of them responsible for bringing TLA+ there-- I asked him how the engineers found using TLA+, and they said something like, "It takes," said, "about a month to get really

familiar with the language and to get comfortable with it," like any old, you know, any new programming language, but what they found hard was learning to abstract away the details from their specification, and slowly, with use, they learned to do that and that ability to abstract improved their designs. And he said that with no prompting. But that was music to my ears, because my hidden agenda is that the model checker will attract the engineers, get them to start using TLA+, but the real benefit, the larger benefit, is that it teaches them to think abstractly and to become better system builders, and other people have-- other engineers have confirmed that that does happen.

Levin: Well, you wrote a book about-- on specifying systems, in which abstraction plays a pretty significant role. Maybe that was even one of your goals in the book was to help people who have trouble with that notion of abstraction.

Lamport: Well, I'm not sure I would even have been able to express it in that way. One of the things that I've come to realize fairly recently is that the reason I won a Turing Award is for my talent at abstraction. I'm just a lot better at it than most people. And the-- if you look at my most important papers, they're not important because I solved some particular problem. They're important because I discovered some particular problem. I was able to abstract from the unimportant details to get at the heart of what the real problem was. So if you look at, for instance, my "Time/Clocks" paper, it was written about 40 years ago, when the Internet didn't exist. Distributed systems were a lot different today than they were now, but that paper is still considered relevant for people who build distributed systems, because it uncovered a fundamental problem and abstracted it from all the whole mass of details that confront people when they sit down to build a distributed system.

Levin: So while I'm sure that what you said before about you having innate ability here is true, one might hope that at least some of this could be taught.

Lamport: Well, I think that it was certainly developed by studying mathematics, because abstraction is what mathematics is all about.

Levin: I'd like to come back to something that you mentioned in passing. Well, maybe not quite in passing, but ask you to amplify a little bit on it: the fact that in the early days, you were-- I don't think you said "hamstrung," but limited by the fact that you were thinking within the context of programming languages when-- and backing off and doing things in the context of mathematics and not getting hung up on programming languages was an important step and a somewhat liberating one. But it flies in the face of what we might call the current educational system, which doesn't seem to train people as well in mathematics as it did perhaps a hundred years ago, so there's a tension there that has to be resolved, with mathematics as the right way to come at specification and thinking about programs. How do we dealt with the fact that people are perhaps not so well-equipped?

Lamport: Well, before I get into philosophical realms, let me tell you about something that I learned not too long ago. The Rosetta spacecraft that until recently was orbiting a comet -- European Space Agency spacecraft -- had inside of it a real-time operating system whose name I'm blocking on at the moment controlling several of the experiments. What I learned recently is, that operating system was designed using TLA+ and there's a book about it describing their specification and the design process, and I wrote to the-- I guess it was the lead author, who was the head of the team that built that operating system and asked him about some comments about TLA+, and in an email to me, he said that as a result of-- well, this system they built was a second or perhaps later version of an operating system they had built before, and he said-- wrote to me and said that as a result of designing it, doing the high level design in TLA+, the new system-- the amount of code in the new system was 10 times less than in the original system. You don't get a factor of 10 reduction in code size by thinking in terms of code.[1]

Levin: That's for sure.

Lamport: He also said, parenthetically -- and I'll try to quote him as accurately as I can -- he said, "We learned the brainwashing effect of 10 years of C programming." And thinking-- another example, the Paxos algorithm that we discussed. I couldn't have discovered the Paxos algorithm if I had been thinking in terms of coding it, and engineers wouldn't come up with-- today's engineers would not have been able to come up with the Paxos algorithm because they're thinking in terms of code. And you need to think at a higher level. Well, Paxos doesn't solve all distributed systems problems, and in fact Paxos is itself is-- one reason it's so successful is that it's so abstract and so general that you won't be able to find one implementation of it that is suitably efficient in all applications, so you're going to have to do some kinds of optimizations. And if you try just doing those optimizations at the code level, your chances of getting them right are pretty slim. You have to be thinking at the Paxos level, at the higher level, above the code level to be able to have a chance of getting things like that correct. And that's true whenever you have a complex situation. People believe that-- a lot of people believe that if something doesn't generate code it's useless. Well, something that generates code is not going to-- if your goal is generating code you don't want to be using TLA+ because it's not going to-- it's going to keep you from generating code: it's going to allow you to generate less code. And I realize something rather important. People think that modern programming languages -- they allow you to abstract better. Well, programming language constructs allow you to hide information. Hiding information is different from abstraction because if you hide what's going on inside of a procedure, in order to understand what you've written, you're going to have to know what's going on inside of that procedure. And typically, that's done by writing some English. But that's not precise, and how can you understand something if you're not writing it precisely. Programming languages do abstract: what they abstract from is the hardware. You don't have to look at the-- at the silicon to understand what your program does, but you do have to look at the stuff that's hidden inside that program

[1] Correction: Shortly after this interview was recorded, Lamport recalled that he had misstated which version of the operating system for the spacecraft was designed using TLA+. He sent an email to Levin in which he wrote: "I said that a Real Time Operating System [RTOS] designed using TLA+ flew on the European Space Agency's Rosetta spacecraft. I just went back over the relevant emails and realized that this was incorrect. The RTOS that was designed using TLA+ was the next version of the one flown on Rosetta. Everything else I said about the RTOS designed using TLA+ was correct--in particular, that its code size was 10 times smaller than that of the previous version. However, it was that previous version that flew on Rosetta."

to understand it. And what the abstraction of TLA+ does-- it's not hiding what goes on inside that procedure. It's being able to express what goes on in-- what that procedure does in a very simple expression, so you can understand it, and-- Shall I give examples about problems that come up with not having precise specifications?

Levin: Certainly.

Lamport: I mean I'm just thinking of one -- thought of it recently -- that bit us, and I think it was in-- produced a bug in TLC. Java has this file class for writing files and it has-- that class has a delete method. It deletes a file. And it says-- it returns true if and only if the delete operation completes successfully. Perfectly clear. What can be ambiguous about that? Do you see anything ambiguous about that?

Levin: Seems clear.

Lamport: Well, what happens if you delete a file that doesn't exist. Does that operation succeed or doesn't it succeed? Well, at least 15 or 20 years ago, two different Java compilers for two different operating systems had two different answers to that question, and imagine-- first of all, try to think about your favorite method that you heard-- you've read about for writing specifications of Java methods and ask yourself would they have forced the specifier to have answered that question, and then, how many hundreds or thousands of unanswered questions are there in the Java language manual? And you get an idea of how really really really really complicated these simple languages like Java are. I mean my question-- proof of that, how much simpler math is -- when people write a semantics of Java what they do is they translate Java into mathematics. Would anybody in their right mind try to translate math into Java? Enough said about programming languages.

Levin: And yet at least on one occasion that I know of -- maybe the only occasion -- you tried to make the writing of mathematics a little bit easier for engineers by giving them, if you will, a bit of surface syntax that made it look more like programming.

Lamport: Yeah, you're talking about the PlusCal language.,,

Levin: Yes.

Lamport: …which looks basically like a real simple toy programming language except that the language of expressions is any TLA+ expression, which means it's more expressive and more powerful than anything any programming language designer would ever dream of writing. I mean you can very trivially write an algorithm -- let's not call it an algorithm language, since it's not a programming language -- you can run a model checker on it; it gets translated into TLA+, and you can use the TLA+ tools to check it.

But because the expression language is so simple, it's very easy to write an algorithm that says if Goldbach's conjecture is true, do this, else do that. <laughs> Very easy to express. Of course, the tools-- well, the tools wouldn't actually have difficulty checking that because when you have a-- it's very-- you'll often write specifications in TLA+ in which some number can be any natural-- and some variable can have as a value any natural number. And the way you model-check that is you can create a model-- or one way to do it is you can create a model for the-- for TLC that among other things tells it to substitute some finite set, like the numbers from zero to ten for the set NAT of natural numbers. And so you could, in fact, model-check that algorithm, and not surprisingly for the numbers from zero to ten it's going to find that Goldbach's conjecture is true.

Levin: So PlusCal was a little bit of a step in this direction but not really.

Lamport: Yeah but-- well, but, no, it is a step in the sense that I wrote it because Keith Marzullo and I were-- well, we were preparing a course on concurrency, which he actually gave-- he taught once or twice I forget at UC San Diego, in which he wanted to use-- write his specs of his algorithms in TLA+ but he wanted-- instead of using TLA+, something like a toy language that could be translated into TLA+, and so I designed PlusCal and Keith and I together wrote the PlusCal-to-TLA+ translator. And the nice thing about that is that in addition to it being nice-- a nice introduction to TLA+ is that I could publish algorithms in PlusCal and computer scientists could live-- could deal with it because basically the necessary PlusCal constructs they needed to understand it could be explained in a couple of sentences. So like I said, if I call it it's-- PlusCal you should think of as a substitute replacement for pseudocode. It makes pseudocode obsolete. It has almost all the nice properties of pseudocode in terms of its expressiveness, but you can actually check your algorithms. So that's how that came about. I have conflicting reports about whether one should teach TLA+ by starting from PlusCal and going up, or whether one should start straight in on TLA+. And I think my approach is going to be in the immediate future is for doing it in TLA+ directly.

Levin: I had one more question I wanted to ask you about specification, and it has to do with one of the scourges I would say of our-- of our modern society, mainly malware, and in general, the kinds of security problems that are a daily occurrence in modern computing systems. There's a-- if people made more rigorous use of specification technologies and verification technologies of the sort that you've invented, do you think we'd be in a different place, or could we be in a different place?

Lamport: I'm not an expert on this. My sense is that the kinds of holes that hackers find are very low down in the-- in the food chain at the code level, and the-- so the tools-- TLA+ is not going to be-- you're not going to be able to say "Oh, let's just apply-- take TLA+ and apply this to this code we've got, and then we'll find our problems." What I think can happen-- well, first of all, if you can reduce your code size by a factor of ten then you've one tenth as many places for hackers to be able to find. I shouldn't give the impression that use TLA+ and you're going to reduce your code size by a factor of ten. I have no reason to believe that that's a normal situation. That was just one data point. But what I do believe is that, by learning to think above the code level, you will improve your code and make it less likely that you'll leave

holes for your attacker. And I should say: TLA+ is a formal language. I do some programming and I would say in ten or fifteen years I have perhaps used TLA+ or PlusCal maybe a half dozen times on some method that I'm writing, a program in Java. But every method that I write, I write a specification for. Most of the time it's in English. Sometimes with mathematical formulas if appropriate, but I write a specification of what this method is supposed to do -- for two reasons. First of all, whoever is going to have to-- if anyone ever has to modify that code is going to be grateful for knowing what it does, and that person might be me six months later, it often is. But more important is that unless I write down what this method does, I don't know if I really understand what it does. And so this whole way of thinking improves my coding, and it's going to improve anyone's coding who makes the effort of learning TLA+ to write TLA+ specs in a class or something, even if he never-- he gets a coding job and he never writes a TLA+ spec again in his life, he or she will use it, without realizing it, to be writing better code, and I'm afraid that with our coding culture there's no easy solution to the problem of-- the security problem.

Levin: I fear you're right. What you were just saying led me to think about what's actually the next topic I wanted to discuss, which is a different kind of thread that I think has run through your work for decades again, and that is a desire to, and an interest in, communicating clearly, and I'm meaning communication in a broad sense here. If I look at your list of papers, there are a number in there that talk about how to do things clearly: clear presentations or clear writing. The fact that you spend the effort to design ways of, in a structured and hierarchical way, present the mathematics that's part of your proof systems and in TLA and so on, seems to me another example. Tools that you've built -- perhaps LaTeX, the most obvious one as a-- as a tool for helping people to communicate by not getting caught up in so much of the detail of form, and I can go on. But it seems to me there's a lot of-- a lot of that that runs through your work. Do you have any thoughts on this drive to communicate clearly that seems to be underlying a lot of your work?

Lamport: I don't feel it as a drive to communicate clearly. I would say I have a drive to think clearly and-- I also happen to be a good technical writer. I'd say good writer, but compared to real writers I'm a rank amateur, but technical writing is good. I'm a good technical writer, and part of that, I just think, comes from being a clear thinker, and a lot of that is again tied into my talent and desire for abstraction, because I think if you look at what LaTeX is, it's abstracted away a lot of the details of TeX so that using LaTeX one thinks about at-- a higher level above the typesetting level and at the document level. That's the goal. I don't think that was written to-- I mean that was some unconscious urge to get people to communicate better, and I think the-- I mean that-- I have a little one-and-a page note called, "How to Present a Paper," which I wrote mid-seventies when I just got tired of the poor presentations I was seeing people to give at conferences. What was the other one that you mentioned?

Levin: Well, I was thinking about the hierarchical proof methodology and structuring of formulas and so on as another example of how to present things clearer. You talked about the literary style of mathematics, and this is in some sense a pretty big step away from that as a way of being much clearer about what's going on.

Lamport: Well, I think much of my work is not-- is not telling people how-- showing people how to communicate more clearly, but how to think more clearly. So how to write a proof: it's giving a structure, a logical structure to proofs, making them more rigorous and allowing clearer thinking by making it easier to find mistakes. But I don't think of myself as promoting or even practicing clear communication that well. In fact, I think a problem that I've had for much of my career is not being able to communicate well because I have not understood what goes on in the way other people think. I mean, for example, I think a real disaster was the original Paxos paper.

Levin: I was just thinking about that.

Lamport: I mean as I probably mentioned the-- one of the jokes in there was that the paper illustrates things by little scenarios involving people and the people were given pseudo-Greek names, which if you pronounce them out you would figure-- be able to figure out who the computer scientist was I was talking about, but I figure that even people who wouldn't do that would still be able to pattern-match, and I was just surprised by how much that threw readers.

Levin: Interesting.

Lamport: I think it's probably something that dates back to childhood that I never in some sense considered myself really smart. But I did notice that other kids seem to have an awful hard time understanding things.

Levin: If I adjust what I was saying, it's not about clarity in communication, it's about clarity in thinking, which you've been trying to do, as you said, and that that underlies quite a bit of the work, not just in the specification area although that's an-- one evident place for it, but in the way you've talked about, or abstracted, the work in other domains as well.

Levin: Okay.

Levin: Let me throw a quote at you from you.

Lamport: <laughs> I didn't do it. I deny having said it.

<laughter>

Levin: And I took this from something on your webpage from 2002. You wrote in an email interview to someone, "I think my most important contribution is my work on writing structured proofs, which is

perhaps more relevant to mathematicians than computer scientists. In the unlikely event that I'm remembered 100 years from now I expect, it will be for that." So my question to you is-- that was written about 15 years ago-- let me ask it this way, would you care to revise anything you said?

Lamport: Well, I would simply hold that as evidence at my inability to predict the future because even, what is it, fifteen years later it sounds a lot less plausible than it did at the time I said it. I think it should be my major contribution, but mathematicians seem to think otherwise.

Levin: Do you think that's because they don't just even know about it? That it's in some sense in the "wrong literature" for mathematicians? Or is it-- is it the persistence of the more literary form of proving things that they get taught from an early age?

Lamport: Well, the mathematical community didn't seem to have any trouble discovering LaTeX.

Levin: Yes

Lamport: So things that appeal to them they get-- they spread fast. So it's true that not many people have heard of my structured proof method, but that's because people who have heard about it haven't told other people about it and haven't tried doing it themselves.

Levin: I guess we could speculate on why that might be.

Lamport: Oh, I can--

Levin: I certainly don't have any facts.

Lamport: I think the answer is really simple: it's the psychology is all wrong, because what I'm telling people to do is take theorems that they already believed to be correct and write-- go to an effort, the effort of writing a more careful proof to find out whether they really are correct. So in the best case scenario, what they're doing is all this extra work to verify something they already believe; in the worst case scenario, they're going to go through all this extra work and discover they were wrong and they have this theorem has vanished, and the referees wouldn't have discovered the error if they convinced themselves about-- if you-- they were able to convince themselves. So it's just a lose-lose situation for them. You just have to be really compulsive about not wanting to make a mistake. And I guess another aspect of my career has been that I'm really compulsive about not making mistakes.

Levin: I want to come back to that interview that you did by e-mail, again, because another comment that you made there seems relevant to what you've just said. The question that was asked was, "What is the importance of specifications and verifications to industry today and how do you think they will evolve over time? For example, do you believe that more automated tools will help improve the state of the art?" And you answered that by saying, "There is a race between the increasing complexity of systems we build and our ability to develop intellectual tools for understand that complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought." And your interviewer followed that up by saying, "Are you hinting at advances in the deployment of artificial intelligence techniques?" And you said, "Having to resort to traditional methods of artificial intelligence will be a sign that we've lost the race." That was, again, 15 years ago, and the role of thinking what artificial intelligence is and how it relates to thinking seems like an interesting topic to reflect upon. How do you think about that now?

Lamport: I've given a little bit of thought to that in recent months. One thing I've come to realize is that some considerable fraction of the programming that's now done, 80 percent, 97 percent, I don't know, will not be done by people: it's going to be done by computers. When this happens-- again, I-- 5 years, 30 years, I don't know-- I think most of it will be done with the current machine learning techniques. And that's going to produce stuff that is probably better than current programs in that the machine may wind up with an imperfect understanding of what it is that the person who's giving it the task wants done. But at least, that understanding will be implemented in a bug-free fashion and might even succeed in-- I'm not sure if it-- if that will solve the malware problem or else have a situation in which the producers of the software are in no better place than the malefactors of discovering whether or not <laughs> there are holes in it. On the other hand, that's not going to be a satisfactory solution, what I would consider a satisfactory solution, to the problem of getting really reliable software. We really care that-- not that it just works right most of the time, but that it works right essentially all the time. And that's going to be-- I don't think you'll be able to get that with machine learning because we don't understand how machine learning programs work. So if we don't understand how they work, how can we understand really what they're going to do? As long as-- at the moment, we're-- they just happen to be better than most human beings. That's not a problem. So I see two possible scenarios. One is that the use of sophisticated techniques, like machine learning or something, will allow translation from precise mathematical descriptions of what's supposed to be done to the software. In other words, the-- what is now the above-the-code level will become the code level by essentially-- basically will-- machine learning may replace conventional compiling techniques to provide a really higher level of abstraction. The other possible scenario-- no. If that scenario was true, that means that instead of learning C++, people will be doing-- <laughs> for the-- waiting for the future to do a lot better learning, TLA+. <laughs> The other scenario is that those same machine language techniques that we don't understand are going to be used to produce software that we should be understanding, and I don't think that's going to be a pretty scenario. <laughs> But, that doesn't mean it won't happen. <laughs>

Levin: So you don't think there's a fundamental, let's call it, conflict between specification of what something ought to do and machine learning as a technology for getting computers to do something? The challenge is, perhaps, what it means to be able to say with sufficient precision what a machine learning program is-- when it is getting the correct answer.

Lamport: Well-- I don't know enough about machine learning to say anything intelligent about it. But-- if you start-- I mean, what machine learning seems to be really good at these days is taking somewhat vague tasks and doing as good or a better a job as a human-- a human being can do at translating those vague tasks into something that people are happy with. I don't think it's been applied to tasks in which what is to be done is precisely specified and the how-it-gets-done is left to the program. Hopefully, one we'll be able to come up with ways in which that can be done sufficiently efficiently and sufficiently accurately to produce the desired result. Whether that is to make the probability of code being incorrect, say, at least as small as it is if you use current code verification-- I mean, formal code verification techniques.

Levin: Mm-hmm.

Lamport: I don't know how it's going to be done, but I don't know how machine learning is really done nowadays. So-- <laughs>

Levin: Well, does any of this relate to what is sometimes called probabilistic programming, meaning that instead of there being an answer, the answer is expressed with some probability or some confidence interval, or something or other like that? Notions that we can make precise mathematically, but which are a little bit different from the right and wrong absolute distinctions that tend to be made in specifying programs?

Lamport: I don't think I-- I'm not-- I would have to guess what probabilistic programming is about from-- based on the word and vague things you're telling me. And I don't think there's much point in trying to do that.

Levin: Okay. All right. Well, we don't have to pursue that, then.

Levin: We've come to a somewhat different topic, although it's suggested maybe a little bit by something you said earlier, having to do with publication, the notion that when you go to publish a paper, of course you have to convince some set of referees and editors that the result is worth publishing. And sometimes, one, you're successful, and sometimes not. You've had this experience along with everybody else who tries to publish. How do you-- if you look at the system of publishing papers and how it has changed or hasn't in the decades that you've been publishing, how well do you think it works and where do we think it's-- where do you think it's going as a tool for communicating scientific work?

Lamport: Well, I can't speak about science. I can only speak about computer science.

Levin: Okay. Computer science.

Lamport: And, in the past, it seems to have worked pretty well. I don't know of any significant work that was inordinately delayed from recognition because of publication-- problems in the publication system. I mean, there have been-- I know of one isolated incident-- instance. I've-- One problem I've-- noticed that-- at least in the field of formal methods, is that-- and I don't know if I mentioned that, that-- you publish something if it's new. If it's not new, you can't publish it, you don't publish it. So if there's some method that works and you said all there is to say about it, there's some tendency for it just to sink out of sight and people will go searching for the literature and will come up with all of these ideas that are worse than that one, but they're the ones that get published. I don't know if this is a significant problem. I mean, I can point to one of my papers, from which I point to that as being an issue, but on the whole, I've been well-served by the publication process. But I've published very little in the last five or ten years, so I really can't say much about what's happening now or what's going to happen in the future.

Levin: How about the potentially transformative effect of self-publishing? You talked about the fact, just now, that people don't find the thing that works, they find the different ideas that don't work. Seems to me that's going to be amplified by the ease with which people can self-publish on the Internet or whatever.

Lamport: I don't know. Perhaps automated things like reference counts will be able to take the place of refereeing. Not an issue that I've given thought to.

Levin: Okay.

Levin: There was actually a-- it seems a few years ago, though, a lot of interesting so-called reputation systems, some which were intended as some way of dealing with the ease of self-publication and the conflict that that poses with more traditional refereed or peer-reviewed or something or other sources that generally have high reputation. But I haven't seen any of those that have actually caught on. Have you seen anything along that line that you would--

Lamport: Well, there's the h-index.

Levin: Yeah. I guess that's probably the main one.

Lamport: Yeah. Which is-- always seemed bizarre to me because you're comparing-- you have some formula that involves comparing time with length and <laughs> thinking that you're getting some meaningful result out of it.

Levin: <laughs>

Lamport: It's-- what is it? The number of papers-- citations is greater than the rank, but why greater than the rank? Why not greater than three times the square root of the rank, or something? There is absolutely no justification for it.

Levin: Right.

Lamport: And I have no idea whether anybody has ever done any study as to whether there is-- you can just simply take that-- those two data points and come up with a more reasonable number than those two values, numbers of citation and-- number of published-- papers published or something. I don't know. I guess it's one of these fundamentally difficult problems that whenever you have any system for judging something, it's all-- it's always a substitute for what you'd really like to be measuring, in some sense, innate quality, and whatever system you use, people will be gaming the system rather than improving the quality. <laughs>

Levin: Indeed. Depressing, but true.

Levin: So I think we've actually come to my last question at this point, which is specifically about the Turing Award. And the Turing Award is not a career achievement award, although sometimes it covers a fairly broad space of work. So I'd like to ask you whether winning the Turing Award has affected the way you think about your work and about the impact that it had? Not necessarily about the work that you do, I'm assuming that you do the work you want to do whether or not you win awards, but has the fact that you received the award affected your-- the way you think about your work, your perspective on your work?

Lamport: That's a tough one. <laughs> Because it-- I mean, in some sense, I think what you're asking for is has it changed a rational assessment that I make? But it's very hard to untangle that from the emotional effects of the award. It-- and sometimes, I think maybe it gets me to take what I've done more seriously and sometimes, I think it doesn't. <laughs>I mean, that-- a lot of the hyperbole that was appearing on, especially, Microsoft websites of when I won the award, credits-- gives me credit for the development of the Internet. And, if I were to really believe that, I would get really depressed, given that the--

Levin: <laughs>

Lamport: --effect-- the negative effects that the Internet has had. But then, good sense comes in and I'd say, well, that was really hyperbole. I had nothing to do with <laughs> the existence of the-- how the Internet turned out. So-- I guess the answer is that-- if you would have asked me that question before I won the Turing Award, what has my work been all about or what good has it been-- I would have been very confused and, having won the Turing Award doesn't make me less confused. <laughs>

Levin: Well, if you think back-- let me push on this just a little bit more-- if you think back to the year following the announcement that you had won the award when you were going around giving talks and this and that, can you recall any incidents that occurred during that time as result of the fact that you were touring as the current award winner that stand out, maybe influenced your thinking about it?

Lamport: I'm afraid that I have to give a disappointing answer of “no”. I mean, before I won the award-- I would get sort of rock star treatment. If I went to give a talk, people would want to take selfies with me to an extent that-- I guess I always felt a little uncomfortable about that because-- I mean, people, they-- the-usually students are reacting to me as if I think I would have reacted to, for example, two of the speakers that were-- that came to visit MIT when I was there, were Niels Bohr and T.S. Eliot. And, I mean, it seems to me that-- and I get the sense that they're regarding me the way I regarded Niels Bohr and T.S. Eliot and I said, "There's something wrong there."

<laughter>

Lamport: But, maybe the number of <laughs> students who were coming up has increased some, but I haven't felt that to have changed significantly. I think I've gotten invitations to talk from people who would not have given me an invitation to speak otherwise, and those, I've politely declined. <laughs> But I'm afraid perhaps I was too old by the time I won it--

Levin: <laughs>

Lamport: --for it to have made the kind of difference that you seem to be looking for.

Levin: Just curious. I'm not looking for anything in particular. I just wanted to know.

Levin: We've pretty much come to the end of my list of questions. But obviously, there are many things in your career that we have not had the time to touch on. If there are any that you would particularly like to highlight, I think this would be a good time to bring them up.

Lamport: Well, actually, I should probably answer a question that maybe you thought you were asking, but I don't think you were, which is that one effect that winning the award did have on me is it got me to look back at my career in ways that I hadn't. And I think it made me realize the debt that I owed to other computer scientists that I hadn't realized before. For example, when I look back at Dijkstra's mutual exclusion paper: now, I've recognized for decades what an amazing paper that was in terms of the insight that he showed, both in recognizing the problem and in stating it so precisely and so accurately. But one thing that I didn't realize, I think, until fairly recently, is how much of the whole way of looking at the idea of proving <laughs> something about an algorithm was new in that paper. He was assuming a-- an underlying model of computation that I somehow accepted as being quite natural. And I don't think I understood until recently how much he created that. And I think there are some other instances like that, where I absorbed things from other people without realizing it. And one of the reasons for that may be that I never had a computer science mentor. I think I mentioned the mentor I had at Con Edison, but that was in programming and sort of-- somewhat of-- some bit of intellectual mentoring, but I never got that in school really. I never had a one-on-one relationship with any professor. <laughs> And so, the whole concept of mentoring is somewhat alien to me. I hope that in the couple of occasions where I've been in a position to mentor others, I didn't do too bad a job, but I have no idea whether I did or not. But my mentors have been my colleagues and the-- I just learned a lot from them by osmosis that, in retrospect, I'm very grateful for, but I was unaware at the time.

END OF THE INTERVIEW

