An Interview with Leslie Lamport
2013 recipient of the ACM Turing Award

Interviewed by:
Roy Levin

Recorded August 12, 2016
Mountain View, CA

This interview was a joint production between the ACM and the Computer History Museum and we think the Computer History Museum for their cooperation in its production

Levin: My name is Roy Levin. Today is August 12th, 2016, and I'm at the Computer History Museum in Mountain View, California, where I'll be interviewing Leslie Lamport for the AMC's Turing Award Winners project. Morning, Leslie.

Lamport: Good morning.

Levin: I want to start with the earliest time that you can remember being aware of computing.

Lamport: Aware of computing. Probably in my junior year in high school. I think I came across a book that discussed Boolean algebra, and must've discussed computer circuitry a little bit, because I got interested in trying to build a computer. Didn't get very far, but I think it was my junior year in high school, possibly senior year.

Levin: So that would've been what year?

Lamport: '55, '56.

Levin: And you were not so much aware of the state of computing devices at that time, rather more the mathematics behind computing circuitry.

Lamport: Well, I mean, I must've been aware of the existence of computers. I seemed to recall they existed in the movies. Wasn't it "The Day the Earth Stood Still"?

Levin: Yes.

Lamport: There was a computer. Or am I just making that up?

Levin: The robot. <laughs> How the robot worked was another matter.

Lamport: Yes.

Levin: I think-- yes. First time computers were-- sort of burst upon the public visibility might've been the election in which they were used for the first time to make predictions. Was that 1960?

Lamport: I don't know. There was a period of time probably when I was an undergraduate and going through large amount of graduate school where I was unaware of television, so…

Levin: So let's, in fact, talk about what happened after you became aware of these things in high school. Did you explore computing through high school and into your undergraduate time?

Lamport: Well, I tried to build a little computer, and this was while I was in high school. And I visited the IBM headquarters in New York, and they were very helpful for this, to this little kid who wanted to build a computer and they gave me a number of vacuum tubes. In those days, they would replace them on regular intervals, and so they were still good. And so I think I got to the point with a friend of building a 4-bit counter. <laughs>And-- But that impressed the interviewers when I applied to MIT. <laughs> So that had a good effect. But my first serious exposure to computers came the summer after I graduated from high school. Through some kind of city program that I forget, I wound up having a summer job at Con Edison, the New York electric utility. And-- that's New York City electrical utility. <laughs> -- and I was assigned to do some very boring stuff. But I learned that there was a computer there and I went up to the computer center and managed to get myself transferred to a job running tapes on their computer system — for the younger generation, tapes used to be a storage medium for computers — and they would be read on these large tape drives and they would also serve as input and output, and so you'd have to mount the input tape for a program, run the program, and then remove the output tape and give it to whoever wanted the results. So that was my job, mounting tapes, probably starting the programs and stuff like that. But in my spare time, and in the computer's spare time, I learned to program it. And the first program I can remember writing computed "e" to something like 125 decimal digits. That number being that the computer I was running on, which was an IBM 705, had an accumulator of 125 or so digits. And so I was able to do that basically without having to do multi-word<laughs> arithmetic.

Levin: So I guess it was natural that your first program involved an interesting mathematical entity, since your interest in mathematics from-- I gather from your resume -- goes back even further than your interest in computing.

Lamport: Yes. I guess in some sense to the extent that an elementary school student could be a <laughs> mathematician, I was a mathematician throughout my school years, even though the idea of actually being a mathematician as a profession never occurred to me until around my senior year in high school. But even so, it seemed like not a practical way of making a living, and so I started out being a physics major. And maybe my first or second year I switched to math. Although I'm not sure if the reason was that I really thought that mathematics was a career move, but at MIT in those days, mathematics was the only major that didn't require you to write an undergraduate thesis.

Levin: <laughs> Practical. As always. So you actually were thinking at some level about careers coming out of high school and going into your undergraduate days, it sounds like?

Lamport: Not in a very concrete sense. In terms of, “Oh. I think if I become a physicist I will be able to get a job doing thus and such,” although it was clear that becoming a mathematician, the only thing I knew that mathematicians did was teach math. So I suppose at the time I switched to a math major in college my career goal would’ve been to be a professor of mathematics. I should mention that throughout my studies, I had part-time and summer job as programming. My undergraduate years at Con Edison. I went back to them for summers, this time as a programmer. And continued. And also worked part-time in the computer lab at MIT, and that was also running tapes. But as an undergraduate, I also remember doing a, working as a programmer, for someone in the business school.

Levin: So programming and math and physics all were interwoven during your undergraduate days, but you thought of them as sort of separate activities in some way.

Lamport: Well, they were interwoven temporally, but they were completely different worlds as far as I was concerned.

Levin: Mm-hm. And when did you first notice or become aware that there might be some — let’s call it science — related to computing as distinct from it just being a technology for calculation or something?

Lamport: That’s a difficult question. I really can’t answer that. There was some point in my career at which I suddenly realized that I was a computer scientist. But I can’t say when — it probably happened by before I went to SRI, which was in 1977. The first thing I published was a letter to Communications of the ACM commenting on an algorithm that had been published. It was an algorithm for implementing a hash table. And I discovered some very slight improvement one can make. Then thinking about the problem, I realized that there was a simpler way to do the hashing and started collecting data on efficiency, or something or other. And when I was in the midst of doing that, someone published a paper in CACM with that method. And for quite a while that was the standard method for implementing hash tables. But I had discovered I think three variants or techniques for implementing the idea that weren’t mentioned in that paper, and I wrote another, a paper just describing those three methods, and it was returned by the editor saying it was just not interesting enough to be worth publishing. In later years, two of those methods, each was published in a separate paper <laughs>in the CACM, but by a different editor. Just an amusing little historical footnote. <laughs>

Levin: Well, you began to learn pretty early on how the vagaries of editing and review in publication work. That’s a topic actually I want to come back to later. But I want to continue now with a more chronological view of things. So while you were at-- you mentioned SRI, but before you went to SRI you were at Compass for some years. How did you come to be working there?

Lamport: When I went back to graduate school to finish my degree, I needed to get a job to support myself. And somebody, actually, a professor at Brandeis, where I was a graduate student, recommended I check with Compass, which, official name was Massachusetts Computer Associates.

Levin: Yes.

Lamport: And they gave me a job<laughs> that, well, first was supporting me through the rest of my graduate studies. And when I got my degree, I was all set to take a teaching job at the University of Colorado in Colorado Springs. And Compass said, "Wait. Would you like to work, keep working for us in the new California office we are going to set up?" And I thought about it and decided, "Yeah, that sounded good." And so by the time I had actually gotten around to being ready to go to California, they said, "Whoops. The California office has been delayed, but that's okay. You can go out to California and work for us." Which I did. And I worked for them for another five years, I think. And they never did build, <laughs>-- open, a California office, but I kept working for them from California and would visit the home office outside of Boston for about a month once a year.

Levin: So that's very interesting: an early example of telework, which is, of course, these days a more common notion. What kind of work were you doing? Maybe it would be useful to talk a little bit about the kinds of projects that Compass did and what the business was.

Lamport: Well, Compass's main business was building FORTRAN compilers. But they also had random contracts with the Defense Department for doing different things. And with some companies. One of the things I did was design a file system for a computer that was being built by Foxboro, and Compass had the contract for doing software for that new computer — I don't remember exactly what. That was, I suppose-- would've been the first concurrent program I actually worked on, if the idea of concurrent programs was known. <laughs> But instead, it was-- actually worked, by using interrupts.

Levin: So I'm intrigued by the remote working arrangement that, in those days, it seems to me, would've been quite a difficult way to develop software. Can you say more about how you actually did that?

Lamport: Oh, I'm sorry. The-- that Foxboro project, was while I was still living in Massachusetts.

Levin: I see.

Lamport: In the course of-- another contract that Compass had was building the FORTRAN compiler for the Illiac IV computer, which was a, an array computer we call today, with 64 processors, which was an enormous number in those days. And I worked on the problem of compiling sequential FORTRAN code into code for parallel computer, in particular, parallelizing FORTRAN do loops. And I developed I guess

you-- an algorithm or I would say perhaps even more so developed the theory behind it, which in terms of theory was pretty simple. It was basically integer linear algebra. And I wrote a-- well, let me start again. At the time, I had a friend who had some land in New Mexico that we would spend-- take a week or two vacation, and stay with them. And what I proposed to Compass was that I go spend — I forget whether it was one or two months there — working on this problem. And they said, "Okay." I did that. And while I was there I wrote a tome, <laughs> a-- I guess you'd call it a paper. Although it was handwritten. Maybe about this thick <laughs> in sheets, and presented it to them. And this I think blew their minds: the notion of using linear algebra. And I learned later from observation that this tome that I delivered, it was like Moses delivering the stone tablets. <laughs> It was practically a sacred text that people studied and they did use it to build the compiler. But what it did serve is to reassure the people at Compass that I could go off by myself without supervision and actually do something useful. So they sent me off to California, and I don't remember if I actually-- what my actual assignments were, if I even had any, but I was being paid on some government contract. And it was I think about six months after I moved to California that I developed the bakery algorithm. And I think from that point it became clear to Compass that I was a researcher that they were going to support. And my memory is that I was left pretty much to myself to figure out what I was going to do and they would find contracts to support me.

Levin: So the bakery algorithm is obviously a very important milestone in your career, and I want to talk about it quite a bit later. But for the moment, I want to pursue this notion that Compass was — even though they didn't really formally have a research organization, certainly not in California — they were in effect supporting you as a researcher; is that right?

Lamport: To the best of my memory, yes. <laughs>

Levin: Uh-huh. Okay. So it seems like it was a fairly open-minded research environment, if you want to call it that. At least as far as your interests were concerned.

Lamport: Well, I think from Compass's point of view, they were happy to work for anyone who would pay them, and they figured out that the Defense Department would pay them to have me do research, and so that's what they did. <laughs>

Levin: So the bakery algorithm, let's-- you invented that — this is we're talking early '70s now, right?

Lamport: Yes.

Levin: And it got published, if I remember, in '74; is that correct? Something like that?

Lamport: That sounds about right.

Levin: Yes. So they were also, Compass was also, comfortable with the notion of independent publication in those times. That you could go off and do this publication presumably on your own. Is that okay?

Lamport: Oh, sure. This notion of intellectual property didn't really exist as a practical sense in the computer industry, at least in terms of any kind of research in those days. I think that it was probably even before the days of software patents. I'm not sure of that. Certainly it would've been before the days of patenting an algorithm. And so, no: there was no notion that anything would be not published. And, of course, anything was published was effectively serving as an advertisement for Compass, so they were quite happy with that.

Levin: In fact, wasn't it, in those times, that algorithms more or less routinely got published in the Communications of the ACM as sort of people throwing out, "Here's my algorithm."

Lamport: There was an algorithm section in CACM. That's not where I published, because those algorithms, they weren't algorithms. It was probably somewhere between algorithms and code, because you had to submit a working program. And how could I submit a working program on the bakery algorithm when multiprocessor computers essentially didn't exist?

Levin: So let's talk a little bit more about the bakery algorithm at this point. Can you tell us a little bit about how you came to be interested in the problem?

Lamport: Oh. I can tell you exactly. In 1972, I became a member of the CACM. And I think one of the first-- no, of the ACM, <laughs> and one of the first CACM issues that I received contained a mutual exclusion algorithm. And I looked at that and it seemed to me that that was awfully complicated: there should be a simpler solution. And I decided, "Oh. Here's a very simple solution for two processors." I wrote it up. I sent it to ACM. And they-- editor sent it back saying, "Here is the bug in your algorithm." <laughs> That taught me something.

<laughter>

Lamport: It taught me that concurrency was a difficult problem <laughs> and that it was essential to have proofs of correctness of any algorithm that I wrote. And it-- well, of course, it got me mad at myself for being such an idiot, and determined to solve the problem. And attempting to solve it, I came up with the bakery algorithm. And, of course, I wrote a careful proof — what for those days was a careful proof. And in the course of writing the proof, I discovered that the bakery algorithm had this amazing property that it did not require atomicity of reads and writes. And I understood that was a really cool thing, and I'm not sure at which point it made me realize that I had really solved the mutual exclusion problem, because I had written an algorithm that didn't require any underlying mutual exclusion: something that, there was

one article by Per Brinch-Hansen that said was impossible. <laughs> So it felt nice having solved an impossible problem.

<laughter>

Levin: So you wrote up the algorithm then and submitted it with a proof.

Lamport: Yes.

Levin: Do you recall how the editor of the journal reacted at that point? Was it a straightforward, “Oh, great. I’ll accept it,” or something more complex?

Lamport: It was straightforward. He sent it out to referees. The referees-- I don’t remember any significant comments from the referees. And it was published without a problem.

Levin: So this was piece of work you were doing while you were employed at Compass. Did you share it with colleagues there?

Lamport: I did. In fact, the next time I was out at Compass, I showed it to a colleague, Tolly Holt — Anatol Holt. And I got a reaction I don’t think I’ve ever gotten from anyone else, which told me that Tolly appreciated that algorithm more than anyone else had. Because when I showed him the algorithm, he said, “That couldn’t be right.” And, you know, I showed him the proof and he listened to the proof, he went over it, he said, “Well, I don’t see any hole in this algorithm, but it must be wrong, and I’m going to go home and think about it.” And I’m sure he went home and thought about it, and, of course, never found anything wrong with it. But I was very impressed by this visceral approach that Tolly took to the issue. Tolly, by the way, was, he was an -- I don’t know if you’d call him an advocate — but a believer in Petri nets. And he was also interested in what I would guess we would now say was fundamental issues in concurrency. And every year for about three years, <laughs> I would go back to Compass and Tolly would have his new theory of concurrency, and he would give some lectures on it. And it would start off, in some completely different way, but then it would always wind up at Petri nets.

<laughter>

Lamport: And it was all very lovely — you know, carefully reasoned — but it just struck me as being not very useful, because I realized that Petri nets had nothing to tell me about the bakery algorithm. <laughs>

Levin: Was that a conversation that you had with him about the-- your study of the bakery algorithm and the fact that his methodology for talking about concurrency didn't work with it?

Lamport: No. I don't think we ever had a discussion about that. I learned some, I think, important basic things from Tolly, and I'm not sure what they were, but I think the experience of thinking about things in terms of Petri nets was in fact useful. Tolly is probably best known for-- as being one of the inventors of marked graphs, which are a particular class of Petri nets that are quite interesting — they have very nice, very simple properties, and in fact, are a useful model of a lot of real-world computation. It just-- they weren't going to be useful to me in general or as a general way of looking at concurrency.

Levin: Coming back to the bakery algorithm itself, I've read in one of your perspectives on it that you thought it was the one algorithm that you discovered rather than invented. I'm not sure I understand the distinction.

Lamport: <laughs>

Levin: Can you elaborate on that?

Lamport: I'm not sure I understand the distinction either. But there's just-- there just seemed to be something fundamental about it. And maybe the basic idea of the use of a counter, a sequence of numbers that can in fact increase forever, that you will see elsewhere in things I've done — in the "Time, Clocks" paper, for example, the use of timestamps or that kind of counter, and in the whole state machine approach, in Paxos, there's-- well, in Paxos, there is also a counter that's being used in, not quite in the same way, but it seems like there's some fundamental thing going on there and I don't know what it is exactly, but it's certainly been quite useful to me.

Levin: All those algorithms that you mentioned are things that we'll talk about some more, because I think that's been a very significant thread in your work, starting with the bakery algorithm. You've also said it was the one you were proudest of. Can you say why?

Lamport: No, I can't.

<laughter>

Levin: Well, it doesn't need to have a rational basis, I suppose. So that happened while you were in sort of independent researcher mode.

Lamport: Let me--

Levin: Yes, go ahead.

Lamport: Let me try to answer the question of why I'm proudest of it. I think in other things that I've done, I can look back and see: "this idea developed from something else." Sometimes leading back to a previous idea of mine, very often leading to something somebody else had done. But the bakery algorithm, it just seemed to come out of thin air to me. There was nothing like it that preceded it, so perhaps that's why I'm proudest of it.

Levin: Well, certainly mutual exclusion is an absolutely foundational problem in concurrency. And to have solved it as you did in the bakery algorithm, certainly is something worthy of being proud of. As part of the publication of the bakery algorithm, you've included the proof that you talked about. Was it common for papers around that time in the early '70s to include proofs with algorithms that were published?

Lamport: I don't know about algorithms in general, but mutual exclusion algorithms, which are pretty much the only concurrent algorithms that were published in those early days, approximately, but was the most significant concurrent programming problem. And yes. Dijkstra, Dijkstra's first paper, had a proof. And what's interesting, the second paper, which I believe was a note submitted to CACM, didn't have a proof. Knuth then published the third algorithm, pointing out the fact that the second one was incorrect <laughs> and I'm sure he had a proof as well. And, yeah, the mutual exclusion algorithms always contained a proof.

Levin: So for concurrency, at least, in those days, that was the norm, even if it wasn't the norm perhaps for other kinds of algorithms.

Lamport: Right.

Levin: Okay. And your work evolved from the bakery algorithm quite a bit, but was often driven by what you had learned in studying it, and in fact, the future study of that algorithm, I think you've written, was something you did for a number of years. Did that study lead you into the approaches for verifying correctness that you eventually spent quite a bit of work on?

Lamport: My interest in correctness was parallel to my interest in concurrent algorithms. I should mention, I don't want to make this-- I don't want to give the impression that I was drawn to concurrency by these fundamental problems. The fact of the matter is that they were just really cool puzzles, and when I was at Con Edison, I had someone who took-- a manager, who became sort of a mentor to me. And he

would give me these problems. One problem he posed to me was: in those days, programs were on punch cards. And you would load the punch cards into the hopper, and you'd press a button on the console. And the computer would read the first card and then put that card in memory and execute that piece of memory as a program. So the first card would usually be something that had just enough of a program in it to load the rest of--, a few more cards. So you'd have a decent-sized program that you would then execute to load your program and start that program executing. Well, I remember, he posed the problem to me to write a loader that worked all on a single card. And I worked on that problem and I would present him the solution and he'd say, "Yeah, that's good. Now make your loader also have this property," and went through, you know, few iterations like that. And I was-- I just loved those puzzles. I mean, one thing I'm also proud of is a program I wrote for the IBM 705, which was: when your program crashed, you would take a memory dump and also presumably print out as much as you can of the registers to kind of find out what went wrong. And, of course, starting that debugging process meant putting a card into the-- punch card into the hopper. And so the program I had was, the problem I faced, was, well, if you do that, well, it's going to erase part of what's there, and executing the program is going to erase some of the registers that you want to, whose contents you want to print out. So how can you do that? And there was one register whose only function was as a destination for a move operation. That as you say move something, and you move a piece of data from some location to the location specified by that register. So the problem was how do you read out that register? Because it could put-- you could put something in the register. You know, you could go anywhere in memory, so you have to look through all the memory. But it could also land on top of the program that you were trying to-- that's trying to find it. So I figured out how to write a program that even if the transmittance for the move instruction put this piece of data right, anywhere inside your program, it would still work. <laughs> And you know, I still think that was a really neat solution.

Levin: What did your boss at Con Edison think?

Lamport: I don't remember if-- I must've showed it to somebody, but I don't remember the reaction.

Levin: Because I was wondering if they actually ended up using it. You could take credit for writing a debugger.

Lamport: <laughs> Oh, no. I don't think anyone cared that much about--

<laughter>

Lamport: --that register to worry about it. But the thing about concurrency is that it added a whole new dimension of complexity, and so even the simplest problem became complicated. It's very much like-- just as trying to write a program where, you know, something could transmit a value into the middle of your program, where in concurrency you're trying to write a-- get a process to be able to do something

even though another process could be doing anything else <laughs> that that process might do. So it posed that same kind of difficult puzzle. And that's what attracted me to concurrency more than a desire for fundamental theory. Once I started working on concurrency, then I became interested in what I might call fundamental problems in concurrency. But initially it was just fun puzzles.

Levin: Very interesting. That's great. So let's continue the sort of chronological thread a little bit. You worked at Compass until '77; is that right?

Lamport: Yes.

Levin: When you moved to SRI?

Lamport: Right.

Levin: How did that come about?

Lamport: Oh, for some reason that I never understood, the people at Compass said that it was a nuisance having me in California that-- they said I had to move back to Massachusetts and I didn't want to move back to Massachusetts, so I found a job in the Bay area.

Levin: Mm-hm. And how did you happen on SRI? It was…

Lamport: I knew people at SRI. I think SRI and Xerox PARC were the only places I knew that did computer science research. And I applied both places, and PARC didn't make me an offer, and SRI did.

Levin: And what was SRI's business around that time? What kinds of things were going on there?

Lamport: Well, SRI was very much like Compass in the sense that if you find somebody to pay for it, <laughs> you can do it. But the major project that-- the obvious project for me to work on — and I suppose they had it in mind when they hired me — was the SIFT project, which was to build a fault-tolerant computer system to fly airplanes, essentially. It was a NASA contract. And that was the work that-- the contract that led to the Byzantine Generals' work. So that's what I started working on.

Levin: Let's talk about SIFT a little bit more. So that was already underway when you joined SRI. And it seems like in the late '70s, if we sort of think back to the state of aviation in those times, the notion of

having computers flying an aircraft was pretty audacious. Do you know how it was that that contract came about?

Lamport: Well, I don't think it was such an audacious idea. And the motivation for it, as I understood it, was that was not long after the oil crisis of the early '70s, where because of an embargo — or some kind of embargo, it's not clear exactly what was permitted and what in fact wasn't — there was an oil shortage in the U.S. and lines at gas stations. And that didn't last for very long, but it made people think about trying to save gasoline. And the one way that they were able to save gasoline or jet fuel was designing airplanes — well, designing airplane you want to minimize the friction and a large source of friction are the stabilizers: the horizontal and vertical stabilizers that control the plane and keep it flying straight. And they realized they could cut down friction and save fuel by reducing the sizes of those control surfaces. But doing so would make the plane aerodynamically unstable and un-flyable by a human being. And so only a computer would be able to fly it, because you had to basically make corrections to the control surface on every few milliseconds. And if you didn't, you could get vibrations and the wings would fall off. So I think another motivation was-- another thing that reducing the size of the control surfaces did, was make the plane really maneuverable. So you could build fighter jets that could really maneuver or do tight turns and all that stuff, but if you could get computers to fly them. So I think those were the, NASA's, two motivations.

Levin: Mm-hm. And the project was underway, as you've said, when you joined. What was your role in that project when you joined it?

Lamport: Well, my general role was member of the team, engaging in all the conversations and lots of decisions and stuff. But perhaps the question you're asking is which of the results that are still known, under the aegis of the Byzantine Generals' work, did I contribute to and which were there when I arrived. And the idea of what was there was the notion of the problem, namely achieving consensus in the face of completely unreliable processors. And the solution that did not employ digital signatures was known. The solution for—and also the impossibility, the result: the result that you couldn't do it with three processors, you needed four. And the algorithm for four processors I believe was invented by Rob Shostak before I arrived. And the generalization to 3n+1 processors was done by Marshall Pease. And that was an amazing piece of work of Pease. I don't know how he figured out that algorithm. Nobody could understand it. I could read the proof and follow the proof and was convinced that it was correct, but I still couldn't say that I understood it. And what I contributed to the original paper was the algorithm with digital signatures, which was actually something that I had done before I got to Compass. Before I got to Compass, I wrote a paper, and I think it was published in '77 — and nobody noticed it — <laughs> which actually generalized the "Time, Clocks" paper to the problem of implementing a state machine in the presence of arbitrary failures. And since in those days it wasn't clear what you'd consider as a failure, I just considered arbitrary failures, that is where a failed process-- these days, most work on failures assumes that processes failed by crashing. But it wasn't clearly the right model when-- or the useful model, in those days, so I had an algorithm that would tolerate arbitrary failures, including processors that acted maliciously. And I didn't discover the impossibility result because it doesn't apply if you have digital

signatures: digital signatures you just need a majority of processors to be working correctly. The other thing that I hadn't done is to pull out the consensus problem. Now, it's clear that if you're trying to build a system that works correctly, you have to choose a sequence of operations that that should do. And so you have to form consensus on choosing each of those commands. And I hadn't pulled out the consensus problem as something to be considered separately, as sort of, as a building block. And I'm not sure why, but I think in retrospect it's because the solution to that problem, the one using digital signatures, came so quickly to me that I didn't think much about it. And the reason the digital signature algorithm came from me is in those days hardly anybody even knew about digital signatures. I happened to be a friend of Whitfield Diffie, and-- who, of course, with Marty Hellman, published in around 1976 the seminal paper "New Directions in Cryptography" that introduced public key encryption and started the whole modern field of cryptography. At any rate, it was I think around '74 that I was having coffee at a coffeehouse with Whit and he proposed the problem to me, which-- he hadn't-- he realized it was an important problem and he didn't know how to solve it. And I said, "Oh, that doesn't sound very difficult." And I thought a minute and literally on a napkin I wrote out a solution involving one-way functions. And that was the first solution: Whit and Marty published it in their paper, crediting me, of course. So I was one of the few people in the world who knew about the digital signature problem and knew that it was possible to create digital signatures. Although in the context of fault tolerance, creating digital signatures should be trivial, because you're only worried about a signature being forged as a result of a random failure, not as a result of a malicious processor, so it should be fairly easy to devise some algorithm that would guarantee that the probability of a forged signature is infinitesimal. Although I've never had occasion to really think very hard-- long and hard about that problem, and I don't think anybody else has either, so it's in some sense an unsolved problem of creating an algorithm that you can prove in the absence of a malicious failure has a suitably low probability of being broken by a random failure. At any rate, so that was the situation when I got to SRI: that original paper I contributed the digital signature algorithm. And perhaps the other important thing that I did was that I got them to write the paper, because I realized that this was an important result, an important problem, and publication, especially outside of academia, didn't seem to have as high <laughs> a priority. As a member of Xerox PARC, you may remember that it was known-- there's a lot of great research being done, but not very much publication and it was known as the black hole of computer science. Anyway, I sat down and I wrote a paper on it and I remember Rob Shostak not liking <laughs> the way I wrote the paper at all. And so he sat down and rewrote it and I imagine that something would've gotten published eventually, but I think it certainly got published sooner because I spurred them into action. And in the second paper, the only really new result in there was an understandable version of the general algorithm without digital signatures — a recursive version — and that was, I think, that was one that I discovered. But the real contribution of the second paper was introducing the name "Byzantine Generals". <laughs>

Levin: Which is what it's, of course, become known as ever since.

Lamport: Mm-hm.

Levin: Just to wrap up on SIFT…

Lamport: By the way, you should ask me about the origin of the name at some point.

Levin: I will.

Lamport: Okay.

Levin: Yeah.

Lamport: Yeah.

Levin: Just to wrap up on SIFT, how did that project conclude? Did the system get built? Did it get deployed?

Lamport: A prototype system was built. It was, I think it was run, on a simulator. I'm not sure if it ever flew in an airplane. There was a bit of controversy at the end of the project. The project was to include a complete verification of the code. And there was some controversy about that. And the short story is that through carelessness, the paper that was published about it did not make it perfectly clear what had been verified mechanically — what had been mechanically proved, and what hadn't. I think in the particular case, that the code that had been verified didn't include some optimization that was later made. So the final system included code that had not been mechanically verified. And there was this general sense in the criticism of it that the-- this was this multi-million-dollar project and, you know, they lied about, you know, the work at that time and they lied about what they had did, what they had done. And the fact of the matter was that this project was used somewhat as a cash cow at SRI in our group. And the actual writing — the actual proof of the code — was an onerous task that nobody wanted to do. And I managed to get off the project before <laughs> it became my turn to do that.

Levin: <laughs>

Lamport: And finally, Rob Shostak-- no, hold. No. It was Michael Melliar-Smith and Rob Shostak, in a really Herculean effort, as the deadline was approaching, wrote the proof and ran it through the prover. And it was really a, I would say, a baling wire and chewing gum <laughs> type of project as far as this proof was concerned rather than this, you know, grand effort. And I think that although the people were careless in what they published — this was not a serious offense, and I think the project was quite successful.

Levin: And do you know if it had any direct impact on the way avionics systems were subsequently designed?

Lamport: Well, I don't know how avionics systems are designed. The Byzantine Generals work certainly did. I was actually curious about that, and sometime maybe 10, maybe 15, years afterwards or so, I happened to be contacted by someone who had been at Boeing at the time, and asked them if they knew about the Byzantine Generals' work. And he said, "Yes, indeed." He said he had been the one who first read our paper and he said, "Oh, shit. We need four."

<laughter>

Lamport: Though he did say that this was in the commercial side, and he did say that Boeing had just bought McDonnell Douglas, I believe, who were doing the military aviation side. And that he-- from what he said, at that time, the people at McDonnell Douglas were somewhat clueless about these things. I don't know what goes on these days at Boeing. I have no idea what goes on at Airbus. I happened to visit Airbus and was told that they use a primary/backup system rather than the SIFT-type design, which was I guess what you'd call peer-to-peer these days. And, basically the programmers-- that was the job of the hardware people, that was done in hardware. And you can build a primary/backup system correctly taking, you know, handling Byzantine faults. On the other hand, I've seen occasion where engineers have very happily presented solutions of provably unsolvable problems. So I don't know-- so I'm not positive that the engineers who build the Airbus know about the Byzantine Generals results, so… But Airbus is very secretive about what they do, and you or I can't find out. <laughs>

Levin: You don't let it affect your flying habits.

Lamport: One thing that I've observed is that-- well, let me give you a, one little anecdote. As you'll remember at Digital SRC Lab, where we were, there was a reliable distributed file system that was built called Echo. And that actually motivated me to discover Paxos, because I was convinced that what they were trying to do was impossible and was trying to find an impossibility proof — instead I found an algorithm. But there was no algorithm behind that code: there was just a bunch of code. And I'm sure that there was kind of failure that Paxos would handle that the Echo code didn't handle. But the Echo system ran perfectly reliably. For how many years? I don't know — five years or something like that. And it only crashed once, and the reason for that crash had nothing to do with the algorithms. It had to do with a very low-level programming detail that buffers got overfull. And so whether or not the Airbus engineers will handle all possible Byzantine faults: I think that in practice if they, they're probably good engineers, and they've probably thought about the problem enough that if there are failure modes that would cause their algorithm to fail, they are sufficiently improbable that they're not the most likely cause of an airplane crash.

Levin: So you were at SRI for seven or eight years, and then you moved from an organization that was fundamentally a contract programming or contract project organization into what I think we might call a corporate research lab at Digital Equipment Corporation. How did that move come about?

Lamport: Oh, well, the actual details is that we were in a group that had, I don't know, a dozen people or so. And at some point, they decided that we needed another level of management <laughs>, and that, you know, in addition to the lab director and assistant director, there was now-- 4 of these 12 people became, you know, some managerial title and something, and I just said that was ridiculous. <laughs> And Bob Taylor had just started the SRC Lab, and it was a natural place for me to go.

Levin: So you came into what must've been a fairly small group of researchers at that time, since the lab was just getting started.

Lamport: It didn't strike me as being small, because it was probably larger than the group at SRI where I was, so…

Levin: But there wasn't a single project you were basically being brought into as you had been at SRI?

Lamport: No. It was clear that I was being brought in to-- well, to become a member of the lab. There was much more of a sense of community at SRC than there was at SRI, which may seem strange, because SRC was a much larger place. But as someone said at SRI, that grant proposals are our most important project. <laughs> I'm sorry: Grant proposals are our most important *product*. <laughs> And so there was always this question of you were working on a project and there was no sense of interaction between projects, so in a sense the lab was more splintered than it was at SRC, especially in the beginning where there was one project, which was building the computing environment. So it was clear that my expectation was that I would be continuing the same line of research I had been doing, but also interacting with the other people at the lab, because they were building a concurrent system and so they would have concurrency problems, and so I expected to be useful.

Levin: So how would you characterize the most important product of SRC, by analogy with what SRI said? It clearly wasn't grant proposals.

Lamport: I think-- well, I'm getting out of my depth, because this is a question of why did Digital open the SRC research lab and what did Digital expect to get out of it. Of course, as you know, the kernel of the people who started SRC were the people who came from Xerox PARC, where they had just created personal computing <laughs>, and so there was very much a sense that, well, this was what the lab was going to continue doing: creating the computing of the future. And so that was the purpose of the lab.

Levin: Mm-hm. And so there was a qualitative feel to that organization that was different from SRI; is that right?

Lamport: Definitely. But that, I think, had a lot to do with Bob Taylor, the lab director and lab founder, who was just a wonderful manager. <laughs> And as you know, he was your mentor. <laughs>

Levin: Indeed. Did you consider, when you were looking to leave SRI, other places to go work? Perhaps academia?

Lamport: I did talk to Berkeley and Stanford. But I must say, I was never really serious about it. I did not see myself as an academic, so it was clear to me that SRC was where I was going to go <laughs> if they made me offer.

Levin: Say little bit more about that. What is it about either the difference between SRC and academia that attracted you to SRC more than the university?

Lamport: Well, there were two things at play. First one was personal, and one was institutional, you might say. Personally, I had never taken computer science seriously <laughs> as an academic discipline. And in fact, when I started out, it wasn't. I don't think there was enough stuff you could call computer science, and certainly not in my field, that merited being taught <laughs> at a university. And this may be just my own background which, since I had studied math and physics, and perhaps if I had gone to an engineering-- been an engineering major at MIT, I would've had a very different perspective and I would've considered programming to be a perfectly fine academic discipline, but it never seemed to me that people ought to be going to a university to learn how to program. And by the time I began to feel that, "Well, maybe I actually could go to a university and teach academically respectable things <laughs> to people," I was too set in my ways. And I think that maybe would've been in the '90s or so. So that was one reason why universities didn't interest me. And the other thing is that I found industry to be the best source of problems, and I think in the early days of concurrency, the early days — you know, starting in the late '70s and going-- yeah, it was the late '70s and early '80s that we were talking about — a lot of academic research was just self-stimulated. <laughs> People, especially theoreticians, contemplating their navel and <laughs> deciding what should happen. And I've always found that the real problems, you know, the important problems, were motivated by people in industry having things they wanted to do. SIFT was an example of that. It wasn't sponsored by an industry, but it was very much a real-world project. I mentioned Paxos, which was inspired by a real system being built at SRC. The bakery algorithm: not an industrial problem, but it was clearly important in programming. And what other ones? Oh, Disk Paxos, for example, another algorithm that happened because there was a DEC engineer in Colorado who said he wanted to build a system where instead of having to have three processors, he wanted to have three disks instead to get the redundancy, and that led to Disk Paxos. Fast Paxos — no, sorry, not Fast Paxos — fast mutual exclusion came because the people at WRL, the Western Research Lab, it was another DEC lab, but the people there were building a computer and they needed a mutual exclusion algorithm that had certain properties. So yeah, I've just always found industry to be the best source of problems. Not all problems, but sometimes, I suppose, the most successful problems are ones where you think of the problems before industry realizes they have them. <laughs> But it's certainly my

experience working with industry, and even before that, of programming, has been much more useful than most of what <laughs> I learned in school.

Levin: So since ’85, was it, that you joined SRC, approximately?

Lamport: Yes.

Levin: Until the present day. You’ve been in research labs first at DEC and Compaq and now at Microsoft.

Lamport: Right.

Levin: And that interaction of real-world problems and research investigation has continued through that, essentially, this whole portion of your career; is that fair to say?

Lamport: Yes. It’s sort of an irony and a failure that, although I look for real-world applications or real-world problems, I didn’t interact with the people at SRC that much. And I think that I must have a prickly exterior<laughs> that didn’t encourage people to come and-- come to me with the problems they were doing. And I wasn’t active enough in going out and finding people’s problems, because I always had things of my own that I was working on and it was always easier to stay in my office and keep working on those than try to go out and talk to people and extracting their problems from them. So it more often would come from somebody outside the lab who had-- knew about me or knew about my work and came with a problem.

Levin: You’ve had a lot of external collaboration around problems and their solutions with people in academia, even though you didn’t want to be an academic yourself, so there are evidently at least some people in academia that want to attack real-world problems.

Lamport: I don’t feel like I’ve collaborated with that many people. I think certainly compared to most computer scientists, I think I have more single-author papers. I collaborated quite a bit with Fred Schneider, and I guess that’s because Fred has, I think, a very similar sense of practicality that I do. I collaborated on a couple of papers with Eli Gafni. They both happened while Eli was visiting SRC, and-- well, one of them was the Disk Paxos problem, which as I said, came externally. The other one was on the mailbox problem, and that was a problem that I had had myself, or a problem that had occurred to me many years earlier and just Eli’s being there was an opportunity for it. There was the collaboration with the people at SIFT, at SRI doing SIFT. So that was in some sense the one time where I was really part of a group that I was actively collaborating on doing research.

Levin: So I'd like to shift gears now. And we've done a kind of a somewhat meandering exploration of your career chronologically, or at least part of it. But I think I'd like to start talking about some of the themes that have run through your career — and we've touched on a few of them briefly, but I would like to get into more detail on that. And the natural one to start with is probably the one that you worked on earliest, which we've talked about a bit: the bakery algorithm and concurrency, and concurrency algorithms. You've said that quite a bit of the work you did grew out of the bakery algorithm and studying it, learning from it. One of those was-- well, let me ask you this way. Probably the next famous paper that you produced was the "Time, Clocks, and the Ordering of Events in a Distributed System" paper, which is perhaps the one that's most widely cited. Did that grow out of the bakery algorithm study?

Lamport: Not directly. It had a very specific origin, namely: I received — well, this was while I was working at Compass, and from some connection, I don't remember why, but — I received a technical report written by Bob Thomas and Paul Johnson, which was about replicated databases. And they had an algorithm in there, and when I looked at the algorithm, I realized that it wasn't quite right. And the reason it wasn't quite right was that it permitted a system to do things that, in ways, that seemed to violate causality. That is things that happened before-- one thing that happened before something else would wind up influencing the system as if they had happened in the opposite order. Now, the reason I realized that has to do with my background. And from my interest in physics, I happen to have a very visceral understanding of special relativity. In particular, the four-dimensional space time view of special relativity that was developed by Minkowski in the famous 1908 paper. And I realized that the problems in distributed systems are very much analogous to what's going on in physics because — or in relativity — because in relativity there's no notion of a total ordering of events, because events will-- to different observers, will happen, appear to happen in different order. But there is a notion of causality, of ordering between two events, in which one event precedes another if, for every observer, it will appear that that event preceded the other. And the basic definition is one in which that is that if an event happens to some person-- and the other event happens to another person, then it's possible for a message or a light wave to be sent from the first person, when that event happens, and that it will arrive at the second person before that second event happens. And I realized that there's obvious analog of that in distributed systems which says that one event happens to one processor before it happens at another processor if there was a message or a chain of messages that began at the first processor after this event and reached the second processor before his event. And it's that notion of causality or "before" that was being violated in this algorithm. And I made a fairly simple modification to the algorithm to correct that. And that's the algorithm that was presented in that paper, along with the definition of the partial-ordering relation. Now, I should mention that I'm often-- one of the things that the algorithm used, and that the Johnson and Thomas algorithm used, was attaching timestamps to messages. That is, each process has its local clock and it timestamped a message with the time on that clock before sending it. And people have credited me with my inventing timestamps, which I didn't because I got them from the Johnson and Thomas paper. And what I didn't recognize at the time-- when I received that paper I just assumed that timestamps were a standard technique that people used in distributed computing. This is the first time I'd even thought about the problem of processors communicating by sending messages. And so I didn't point out that timestamps were used in the Johnson and Thomas paper, and so I-- their paper has been

forgotten and it remains as a, citation, one of the four citations in my “Time, Clocks” paper. One of the others being the Minkowski paper and another one being a previous Einstein paper.

<laughter>

Lamport: So at any rate, that’s just a footnote. The other thing I realized, that I believe Johnson and Thomas didn’t realize, is that this algorithm was applicable not just to any-- not just to distributed databases, but to anything. And the way to express “anything“ <laughs> is a state machine. What I introduced in this paper was the notion of describing a system by a state machine. Now, a state machine is something that’s really simple. You start-- it’s something that’s started in an initial state and it then takes steps which change its state and what it can do next can depend on incoming messages, in this case, or from the environment., or it can come from-- or from the current state of the machine determines what it can do next. Now, the ideas of state machines-- well, finite state machines are ancient, dating to the ‘50s or so. But they were all used as finite states, namely that the state had to be a fixed, bounded set, like it’s a certain collection of registers or something. And it was very clear to me and to everybody that finite state machines were not terribly interesting as ways of talking about computation or about problems. But if you remove the restriction about them being finite, then they’re completely general and you can describe what any system is supposed to do as a state machine. And what I realized and said in that paper, that any system would-- if it is a single system, in a sense-- what it’s supposed to do can be described as a state machine and you can implement any state machine with this algorithm, so you can solve any problem. And as the simplest example I could think of, I used a mutual-- a distributed mutual exclusion algorithm. And since distributed computing was a very new idea, this was the first distributed mutual exclusion algorithm, but, you know, I never took that seriously <laughs> as an algorithm. And nor do I take this, the “Time, Clocks” paper’s algorithm as necessarily a-- the solution to all problems, because although whether in principle, you can solve any problem this way, that is, building a system not without the-- without failures. It didn’t deal with failures. It’s not clear that the solution would be efficient, and there could be better solutions. And in fact, I never thought that my distributed mutual algorithm would in any sense be an efficient way of doing mutual exclusion. Well, the result of publishing the paper was that some people thought that it was about the partial ordering of the events, some people thought it was about distributed mutual exclusion, and almost nobody thought it was about state machines! And as a matter of fact, on two separate occasions, when I was discussing that paper with somebody and I said, “the really important thing is state machines,” they said to me, “There’s nothing about state machines in that paper.” And I had to go back and look at the paper to convince myself that I wasn’t going crazy <laughs> and really did mention state machines in that paper. I think over the years, the state machine concept — thanks in part to Fred Schneider talking about state machines — people have gotten the idea. The way I met Fred Schneider is he sent me a paper. I don’t remember what it was about that he had written. He was still-- I think he may have been a grad student or possibly, I think, a young faculty member. And I sent him-- He was working along, somehow, along similar lines and so I sent him a pre-print of the “Time, Clocks” paper, which I think subsumed what he had been doing. But I thought, “You know, this guy’s… This guy’s pretty good.” <laughs> “I should stay in touch with him.” And I was right. <laughs> So…

Levin: So this paper has been very widely cited. Obviously because you did a foundational thing, and whether people really understood that or not, somehow they decided it was the place to point to in their own papers. Do people continue to-- I mean, this paper was a long time ago — almost 40 years now since it appeared. Do people continue to discover the paper and learn things from it?

Lamport: Well, what amazed me is that I recently learned that there are physicists — or at least one physicist — <laughs> who takes the paper very seriously, and says it's very important in developing a theory of the physics of time, and — Now, this boggled my mind. I could understand-- I regard what I did in that paper, at least the partial ordering that I defined, as being very obvious, to me, because I understood special relativity. And so I figured, "Well, it seemed-- it seems really magical to most computer scientists because they don't know any special relativity." But of course physicists understand <laughs> special relativity and they thought it was a seminal paper. And so I— got me thinking, "Why?" And I finally figured out for the physicist — and I confirmed with one — that what my paper did-- the advance that my paper made, was that Minkowski was talking about, in some sense, messages that *could be sent* to define his relation, and I changed that to messages that *actually were sent*. And this seemed to me to be perfectly obvious but to the physicist this was a seminal idea. So I'm-- it seems that things that seem obvious <laughs> to me do not seem obvious to other people. I've never understood quite why the paper is considered so important in computer science. because the algorithm it presents is of no particular interest. It has historical significance, but I think that somehow what people get out of it is a way of thinking about distributed systems that's so natural to me that I consider it obvious, that is not obvious to them, and gets them to be thinking in a different way. This may be related to one difference between me and I think almost all researchers in concurrency —so, theoreticians — is that in most of the research that I've-- most of the papers that people write or people I talk to, they think of something like mutual exclusion, for example, either as a programming problem or as a mathematical problem. I think of it as a physics problem. Mutual exclusion problem is the problem of getting two people <laughs> not to be doing something at the same time. "At the same time" — that's physics. <laughs> And I look at things from a very physical perspective and-- which I think gets me looking at the essence of problems and not getting confused by language the way a lot of people do. I've observed something in computer scientists that I call the Whorfian syndrome. The Whorfian hypothesis --Whorf was a linguist early in the 20th Century, who-- his hypothesis is that the language we speak influences the way we talk. Well, what I call the Whorfian syndrome is the confusion of language and reality. And people tend to invent a language, and they think that that language is reality. And, for example, if something can't be expressed in the language, then it doesn't exist. Let me give you an example. I've talked about state machines. Well, you can model a program as a state machine, and if you model it mathematically, basically what you're doing is mathematical description of a state machine. And so-- a state machine has to consist of the complete state of the system. Now, if you consider a program, part of the state of that state machine describing the program is what I call the program counter: it's the thing that tells you where in the program you're executing, and which statement that you're executing next. Well, there's no way of talking about the program counter in most program-- modern programming languages, and so to people who are hung up in programming languages, the program counter doesn't exist. And so there are cases when in order to reason about programs, particularly concurrent programs, you have to talk about the program counter. And I've talked to people who say that— well, at least one theoretician, he works in programming

languages, said — “That’s wrong. You simply shouldn’t say it.” Wrong-- well, it was the notion-- the programming language jargon for wrong is “not fully abstract.” But he was saying that my method of reasoning about programs, which involved the program counter, which basically you have to use, must be wrong. And a lot of people have created methods that-- since you have to talk about the program counter, they have ways of getting around it by talking, introducing, other things that can act as proxies for the program counter. So that’s just an example of what I mean by the Whorfian syndrome. And I think there are a lot of people who are-- through the years, have worked on problems that were meaningless outside of the context of the language in which <laughs> they’ve been talking about them. And that has not been my problem, I believe. I may suffer from-- the Whorfian hypothesis may apply to me equally well. But I don’t get hung up in the language I’m talking about. I mean, look at the physical reality that underlies these things. And it’s that way in concurrency problems. I think the reason why things that I’ve done have turned out to be significant years later is not that I have any more foresight than anybody else or no better way than anybody else of predicting what’s going to happen in the future. It’s the fact that if I’m dealing-- I’ve dealt with real problems, problems that have a real physical sense, that are independent of a particular language in which talking about them. And it’s those problems have a good chance of turning out to be useful, whereas problems that are just an artifact of the language you’re talking about are not going to seem very interesting when people stop using <laughs> that language.

Levin: That reminds me of another paper that you did about the same time. Not in the same thread as the “Time, Clocks” or the bakery algorithm, but about a problem that was very real at the time, the glitch.

Lamport: <laughs>

Levin:. Where there it seems a very, again, a very physical thing, enabled you to bring some insight to that that others, that eluded others. Could you talk about that a little bit?

Lamport: Sure. Let me explain the glitch, which is more formally known as the arbiter problem. Imagine that you’re-- well, okay, it’s described in some paper by some--, a housewife. (In those days, housewife was not politically incorrect. <laughs>) And the milk man comes to the front door and the postman comes to the back door and they both knock at about the same time. And you have to decide which one to answer. And so it’s a problem of making that decision. And this is known-- philosophers have known of that problem, in the name of “Buridan’s ass”, where instead of the housewife having to decide whether to go to the front door or the back door it’s you have a donkey who has to decide which of two bales of hay to go to, and he’s equally distant from the two. And it was realized in the early ‘70s that computers act like this housewife or this donkey, and have to make a choice between two different things. The way it happens in computers is that you have a clock that’s inside the computer and you have some event that’s external to the computer — maybe a message comes in or something — and the computer has to decide — it’s doing one thing at a time — has to decide whether it should-- ‘cause the clock pulse tells it to “go to the next instruction” or something, and the external interrupt says to “stop what you’re doing and do something else.” And it turns out that if you don’t design your circuits correctly, what could happen in that

case is, instead of saying either “do A or do B” — which in computer terms means either produce a 1 or a 0 — it produces a one-half. And computers are not very happy when-- computer circuitry is not very happy when they see one-halves and it causes them to do weird things and to get different parts of the computer to do different things and your computer crashes. And they discovered that-- I think it was some DEC engineer, who-- there was a conference that was held on this, or a workshop that was held on this problem — And he mea-culpa’d and said that a computer he designed would crash about every couple of weeks because of this problem. And it turns out, this problem is in a sense unsolvable. What’s unsolvable is that it’s impossible to-- if you’re going to come out-- you can come out with a 0 or a 1, but you can’t do it in a bounded length of time, and — which means the computer is screwed because it has to do it by the next clock cycle — and if you do it with a bounded length of time, you have a possibility of getting the one-half. So what computers do-- but you can build circuits that will almost always do it pretty quickly, and as a matter of fact, that the probability of not having made the decision within a certain length of time decreases exponentially with the time. So in practice, you just wait a little while and the chances of getting a one-half are negligible. So that’s what computers do: they don’t try to decide whether this event happened at the same time as the current clock pulse, but whether it happened before the clock pulsed three ti-- you know, <laughs> three pulses later or something like that. And so engineers who know about the problem, they design the circuits. That’s not an issue. Engineers who don’t know about the problem designs the circuits that will fail because of it. And I hope that these days all engineers <laughs> know about the problem. But if an engineer doesn’t know about the problem, he’ll be able, or she will be able, to solve it. You know, they’ll give you a circuit that they say solves it, and of course, it doesn’t. It just pushes the problem to a different part of the circuit. So that’s the glitch or the arbiter problem. And when I wrote the bakery algorithm, someone at Compass pointed out this problem to me and said, “Well, your bakery algorithm doesn’t require atomic operations, but it still requires solving this arbiter problem.” And that’s what got me interested in the arbiter.

Lamport: Should I go into the history of the papers and the publication and stuff, non-publication or not?

Levin: Well, I’m interested in what the impact was of the work that you did and that you published on the arbiter problem.

Lamport: Okay. What I did is I mentioned the--I told the problem, to Dick Palais, who was my de facto thesis advisor at Brandeis and has become a friend. And he realized that an argument saying, against— or the reason that people think that you should be able to solve the arbiter problem is that the argument that you can’t is based on the fact that real world is continuous. And people will argue, “Well, the real world isn’t continuous, because you can actually build this operator that, arbiter, that goes zero to one. Well, it doesn’t work. I can’t-- we don’t know how to do it in bounded time, but still it introduces discontinuity.” And what Dick realized, that if you have the right definition of continuity, in fact, it is continuous. And theorems that tell you that, well, basically theorems that tell you that a system is a solution to a certain kind of differential equation, then it is going to be continuous. And we believe that the world is described by these kinds of differential equations. They are continuous. That result applies, and therefore this is a mathematical proof of the impossibility of building an arbiter that works in a bounded

length of time. So we sent a-- wrote a paper on that, we sent it to some IEEE journal. Might've been Computer. No, not Computer. Might've been Transactions on Computers or something like that. At any rate, the engineers-- this was the oldest mathematics engineers — had no idea what to do about that, and so they rejected the paper. I should mention that this arbiter problem has a long history of being not recognized. And Charlie Molnar, who was a hardware engineer who worked on this problem, said he once submitted a paper on it and the paper got rejected. And one reviewer said, wrote, "I'm an expert on this subject, and if this were a real problem, I would know about it. And since I don't know about it, it can't be a real problem, so the paper should be rejected."

<laughter>

Lamport: Literally. That's the story Charlie tells. I had-- So this paper was rejected from Transactions on Computers. I think by that time the engineers knew about the arbiter problem, but they still didn't know what to do with this paper. Later-- a few years later, a similarly mathematical paper was actually published in Transactions on Computers, but the result wasn't as general as ours. I also wrote a paper called "Buridan's Principle," which basically generalized that not just to computer type of issues but to arbiter-- real-world problems. For example, if you consider, suppose someone is driving up to a railroad crossing, one that just has flashing lights, and the lights start to flash. Well, he has to make the decision of whether to stop or whether to keep going in a bounded length of time, namely before the train gets there. And that's provably impossible. So in principle, there-- it is possible, for him to-- that is, it is impossible to avoid the possibility that he will be sitting on the railroad tracks where the train arrives. And I remembered actually from my youth some train accident that happened where — I think was a school bus — that for no apparent reason, that went through a flashing light and was hit by a train in perfect visibility and nobody could explain it. And it occurred to me that maybe that could be an explanation. I have no idea whether that kind of thing — which is in principle possible — whether it has a high enough probability of being a real problem. And that, that occurs everywhere. And for example, you think of airline pilots have to make similar decisions, and they have to make them quickly. You know, and could accidents actually occur? So I wrote a paper and I wanted to get it to the general-- to the more general scientific community, so I submitted it to Scientific American. No, sorry. Not Scientific-- oh, I forget. No. I think it was Science, the publication of the AAAS. And it was rejected. There were four reviews. They literally ranged from, "This is a really important paper that should be published," to "Surely this paper is a joke." And the two others were somewhat in the mi-- less <laughs> positive or negative. One was somewhat positive and one was somewhat negative, and a 50 percent split was enough to get it rejected from Science. Then I-- usually, when I get a paper rejected, if I think it deserved to be rejected, or if there was a good argument for rejecting it, I will not try to publish again. But I felt that this was worth doing again. So I sent it to Nature, which was sort of the British equivalent of Science. And I got back a letter from the editor who said, who read the paper, he said, he pointed out, "Well, I see this problem." And I realized it was a reasonable problem, in the sense that for someone outside the audience that I'm familiar discussing with. So I described-- kind of the answer to his thing. And then he said, "Okay." You know, and then he sent me something else, another issue, and I said, "Oh, yeah, that's a good point, but here is the way the-- what it should have said to-- with that,"and two or three exchanges like that, and I was never sure whether he was really serious or whether he just thought I was a crackpot and was humoring

me, and then-- but after that exchange I then got an email from some other editor saying that the original editor had been reassigned to something else, and your paper had been assigned to another editor, to me I guess, and I have this question… and he came back with basically one of the questions he had already-- and at that point I just let it drop. Finally, somebody-- the paper was on my website, and somebody suggested I submit it to the "Foundations of Physics" journal, and I did, and that was accepted <laughs> with no problem, but, again, not the wide audience that I was hoping it would get.

Levin: So within computing it sounds like this principle is probably not as widely understood as it should be, still. Perhaps at the level of people who build circuits it's understood, we hope, but perhaps not elsewhere, and these things arise in concurrent systems all the time, right?

Lamport: The people who build computer hardware are aware of it. I don't know whether engineers who aren't principally computer hardware designers know about it. For example, back in the SIFT days I tried to explain this problem to the people who were building the computers that were going to run SIFT, because SIFT had to worry about all possible source of failures, and one possible source of failure was an arbitration failure, and I wanted to ask the engineers whether they understood this problem and what they could tell us about the probability of this happening. Well, in about 30 minutes I was not able to explain the problem to those engineers so that they would understand it and realize that it was a real problem. In those days the avionics people-- computers were just starting to be used in avionics, and the avionics people knew an awful lot about feedback control systems and such like that, knew nothing about computers or the digital domain. So do the people who are building avionics hardware these days know about the arbiter problem and know how to take it into account? I don't know. I sure hope so.

Levin: Agreed. Did any other work come out of your interest in arbitrer-free problems?

Lamport: A couple of things. First of all, I decided-- I mentioned Petri nets, and I mentioned marked graphs. Now, people in the small part of the hardware community that deal with asynchronous circuits and know-- really understand things about arbiters and bui..-- try to build circuits without arbiters — know that marked graphs describe the kind of synchronization or-- the kind of synchronization described by marked graphs can be implemented without an arbiter. And it's significant because the kind of synchronization that marked graphs represent, seem to describe what I would call parallel computing, as distinct from concurrent computing. Parallel computing, in my use of the terms, means a bunch of processors that are built deliberately to collaborate or-- or join together, deliberately to work together on a single thing. Concurrency, is where you have a bunch of processors, sort of, doing their own thing but they involve some synchronization with others, and that, in order to keep from stepping on each other's toes or bad things happening. And, parallelism, when you read about big data computation, where you're using a server farm to compute to-- to process all of the queries to Bing in the last 24 hours — what you're doing is parallel computation. And there are--there's software to do that, I've forgotten the names of the types, Map/Reduce. So for example, Map/Reduce is software that solves the kind of synchronization that is described by marked graphs. At least that's what I presume Map—Map/Reduce does. And, for a

long time, I thought that marked graphs described precisely the kind of synchronization that can be done without an arbiter. And so I decided to-- to try to prove that and write a paper about it. Well, I discovered that I was wrong. That there actually is a weaker type of thing you can describe that-- I'm sorry, a more-- you can describe things that marked graphs can't describe, but that can be done without an arbiter. And, so I wrote a paper about that. And, I wrote another paper that realized that parallel computing is-- is important even though I generally stayed away from it, as not being that interesting. Just problems of-- of competition are much more difficult and interesting to solve than problems of cooperation. But, I just realized that there was an algorithm that had, sort of, in the back of my mind or— that indicated how you could have a-- an implementation of marked graph synchronization by multiple processors that are communicating by shared memory. And so, I wrote a paper about that describing that algorithm. And that's, I think, the extent of what I've done in terms of arbiter problem.

Levin: And roughly when was that work?

Lamport: I believe the work was done since I got to Microsoft, but I think it was the early part of this century.

Levin: So, a decade or so ago.

Lamport: Yeah.

Levin: Yeah. Has this whole area of arbiter-free algorithms, or implementations, or whatever, continued to receive a lot of attention as far as you know?

Lamport: I don't think it's ever received a lot of attention.

Levin: Okay.

Lamport: There's a small community of hardware builders who are-- well, almost all computers these days, or all computers you can go out and buy the store, work by having a clock. And they do something on each clock cycle. So they'll do-- do something, wait for the gates to settle down, and go on to the next step, and that waiting is synchronized by a clock. This is a problem because your computers are so fast that propagating a clock signal from one end of-- of a chip to another takes a significant amount of computing time. But engineers work hard to solve that problem. There's another small band of dedicated hardware builders — Ivan Sutherland is-- is one of them — who believe that it's time to stop using clocks and to build asynchronous circuits. And so, you want to avoid arbitration whenever possible in-- in building asynchronous circuits. They have the advantage of potentially being much faster because if you're clocked, you have to go at the speed of the slowest component or the worst-case speed of the

slowest component, whereas if you're asynchronous, you can have things done as fast as the-- the gates can actually do it. And I know I've heard Chuck Seitz, who's another member of that community who I used to interact with -- used to chat with him a bit and I'm sure learned a fair amount from him — he has described building, basically really, really fast circuits for doing signal processing with this asynchronous technique. But so far it hasn't been adopted in-- in the mainstream.

Levin: So, it sounds like there's at least a potential that you might do more work in this area in the future.

Lamport: I think, my days of working in that area are-- are done. But somebody else might take it up.

Levin: Okay.

Let's pick up the thread of your work in concurrency after that time. You continued to do work in concurrent algorithms and I think there was a paper that you did not terribly long after that. Maybe it appeared around the same time as the Time and Clocks paper that was about concurrent garbage collection that may be the first paper, I believe it was the first one I knew of, where you were working in some sense directly with Dijkstra, although I might have gotten that wrong. Can you talk about that a bit?

Lamport: My involvement that Edsger Dijkstra wrote with some of the people around him — loosely call it his disciples, the authors on the paper; I don't remember whether they were still students at that time or what — about a concurrent garbage collector, and-- he wrote it as an EWD report (the series of notes that Edsger distributed fairly widely) — and he sent me a copy or I received a copy somehow, and I looked at the algorithm and I realized that it could be simplified in a very simple way. Basically it treated the free list in a different way than it treated the rest of the data structure and since it's a concurrent garbage collector I realized that you could just treat it as just another part of the data structure, and moving something from the free list someplace else is the same as— basic problem as making any other change to the list structure. So I sent that suggestion, which to me seemed perfectly trivial, and Edsger thought that it was sufficiently novel and sufficiently interesting that he made me an author as the result of that — something I imagine he regretted doing. <laughter> Yeah, there was an interesting story about that. Edsger wrote a proof of correctness of the algorithm and it was in this prose-style, mathematical-style proof that people used and people still use to write proofs, and I got a letter from Edsger — those days one actually communicated by writing letters and sending them by post — and it was a copy of the letter he sent to the editor withdrawing the paper that had been submitted saying someone had found an error in it. Now I found Dijkstra's proof very convincing and I decided that, oh, there must have been just some minor, insignificant error, and I had just developed the method of reasoning formally about concurrent algorithms that I published in, I think it was, in the '77 paper proving the process of multiprocess programs. So I said, okay, I would sit down and write a correctness proof and I'll discover the error that way and I'm sure it'll be a trivial error, easy to fix. So I started writing a proof using this method that I had developed and in about 15 minutes I discovered the error and it was a serious error. The algorithm was wrong. I had a hunch that it could be fixed by changing the order of two instructions, and I didn't understand wh--, I didn't

know whether that would really work or have any very good idea of whether it would work or not, but I decided to give it a try and of course what I would do is use my method to write a proof, and it was hard work because writing that style of proof you have to find an inductive variant, which — I won't go into now <laughs>what that is — and that's hard and especially at that time I had no practice in doing it. I spent a weekend, and I was actually able to prove the correctness of my version of the algorithm. It turned out that Edsger had found a correction that involved changing the order of two different instructions because it was somewhat more efficient to do it that way, we used his in rewriting the paper, but there I insisted that we give a more rigorous proof, and Edsger didn't want to give the really kind of detailed proof that I believe is necessary for making sure you don't have these little errors in the algorithms, but we compromised and we had— the invariant, I believe, appeared in the paper and not written formally, but written precisely, and so we had a sketch of that proof. And a little while later, David Gries — I think that he wrote that paper by himself not with Susan Owicki — but David Gries basically wrote his version of the exact kind of proof that I had written. It was the same time I was developing my method David and his student, Susan Owicki — or I suppose since it was her thesis it was must've been Susan Owicki and David Gries — <laughs> were developing what was an equivalent method, logically equivalent, but it was more politically correct because — remember that I told you that you needed to reason about the PC in order to write these kinds of proofs, but they didn't reason about the PC: they introduced some way of writing auxiliary variables that would basically capture the difference — but there was another difference in the presentation, and in some sense they were doing it in a—well, this method of reasoning about programs was introduced by Bob Floyd in a paper calling-- Assigning Meanings to Programs. He did it in terms of writing-- using flowcharts and annotating the flowcharts. Now, Tony Hoare developed another way of doing it called Hoare logic in which it's-- in some sense it's a nicer way of doing things for sequential programs, and what Owicki and Gries did was present it in a language that made it look like they were using Hoare's method, but they weren't really using Hoare's method. They were using Floyd's method but in disguise, and as a result — and their method and mine were logically equivalent — their method became well-known and my method, it took about 10 years before people realized that what I was doing was really <laughs> the same thing that they were doing, and it really-- but people were really confused about-- by their method and in fact Edsger wrote an EWD calling my version of the Owicki-Gries method, which he said was to explain it simply, and I thought that-- my reaction was, "God, the poor people who tried to understand it based on his description." <laughter> When you look at it the way I did it it's very obvious what's going on. You see it. It's clear that you're writing an invariant and each step you're checking the invariant, but the global invariant was what was hidden in Owicki-Gries method, and these days-- well, actually I can't say whether people are still using the Owicki-Gries method. They don't write things in terms of flowcharts. There was a paper by Ashcroft which preceded both my paper and the Owicki-Gries paper in which he talked about doing things directly using the global invariant, and I thought that I was making an improvement on that by basically distributing the invariant as an annotation of the program, of the flowchart, and I've since realized that that's dumb: the correct way of doing it is just thinking directly in terms of state machines and essentially reasoning the same way Ed Ashcroft did, but in those days I still thought doing things in terms of programming languages or things that looked like programming languages was a good idea. I've since learned that if you're not writing a program you shouldn't be using a programming language. An algorithm isn't a program.

Levin: So that's a very interesting illustration of the tie-in between your work on algorithms, in particular concurrency algorithms, and the methods that you used that you developed for proving the algorithms correct, which is I think another major thread in your work. The two have been interwoven over, as far as I can tell, most of your career.

Lamport: Yeah, well, in the early days-- and there are lots of papers written about verifying concurrent algorithms — what I think distinguished me from most of them was that I was doing this because I actually had algorithms to verify, and I needed things that worked in practice, and I looked at a lot of these things and said, "I couldn't use that," and even for the little things that I was going to use and let alone try to scale them up to something bigger. David Gries and-- I think is-- well, he has a long background and I think the people who-- I don't know whether he started as a mathematician or an electrical engineer, I think he started as a mathematician — but people from the old days were, in the old days, in the U.S. at any rate, somehow seemed to be more tied to real computing than people were-- most people were especially in Europe back in like in the '70s, and so, you know, their method was, as I said, it's equivalent to mine and in practical terms of whether you could use it made very little difference which one you were using.

Levin: I want to go back to something that you mentioned in passing in conjunction with "Time, Clocks" paper, which was the use of counters, which became an important notion in subsequent work, and in that "Time, Clocks" paper the counters are essentially unbounded and that obviously had practical implications which you explored further in some subsequent work about what you called "registers". Could you tell us a little bit about that?

Lamport: As you mentioned-- actually it's the bakery algorithm.

Levin: Sorry, bakery algorithm, my mistake.

Lamport: <coughs> In the bakery algorithm the-- well, I sort of said it had no precedents. The precedent, what led to the name was something from my childhood where we had a bakery in the neighborhood and they had one of the little ticket servers that you took a ticket and the next number-- you waited for your number to be called (I suppose if I had grown up in a later era it would've been the deli algorithm), <laughs> and that's basically what the bakery algorithm does, except each process invents and basically picks its own number based on numbers that other processors have, and it's possible for numbers to grow without bound if processors keep trying enter the critical section, and so I realized that the algorithm required multiple registers to store a number because you could run out of-- you could overflow a single register, a 32-bit register for example, in a fairly short time. So the reason why the fact that the registers didn't need-- in the bakery algorithm, didn't need to be atomic meant that it was easy to implement them with multiple word registers where only the reading of a single word was atomic, and so at the first glance it sounds like, oh, that solves the problem because it doesn't matter what order you do, reading and writing of the different words it would still work, except that if you weren't careful you'd like to-- well, you'd

like to prove that the numbers, although not bounded, had some practical bound. For example, that the number would never be greater than the total number of times that somebody has been in its critical section, and it was not trivial to implement that, and so the paper I wrote — I think called “On Concurrent Reading and Writing” — gave algorithms for solving that problem, and a nice statement of one of the problems solved in the algorithm was: Suppose you have a clock in your computer, and the clock will count cyclically — you know, say for instance, one day or so: it might cycle from 0 to 24 hours and then back to 0 — but how do you accomplish that if your clock had to be multiple registers, and so the first question is “what does correctness mean?”, and correctness means that if the clock is counting the time, the value that a read returns is the value of time at some point during the interval in which it was reading, and so it’s a nice problem and if anybody hasn’t seen the solution I suggest it as a nice little exercise. So it was the bakery algorithm that led to-- me to look at that class of algorithms. There’s a funny story that the-- it turns out that you don’t have to assume, if you have multiple words, that the reading and writing of an individual word is actually atomic. There’s a weaker condition called regularity that I won’t bother explaining, but it’s weaker than atomicity, and it’s a condition that-- for a single bit it’s very easy to implement in hardware. As a matter of fact, the condition is basically trivial to satisfy for a single bit, and when I published the paper in CACM, I wanted to introduce the definition of a regular register and talk about how they could be used to implement, for example, this kind of cyclical clock, but the editor said that the idea of reading a single bit nonatomically was just too mind-boggling and that he didn’t want to publish a paper with that. So I was forced to state that you had atomic bits even though that wasn’t a requirement. At any rate, that then eventually led me to think about-- actually not eventually, but fairly quickly — thinking about what kind of registers do you have that are weaker than atomic register, and there’s— actually it’s a weaker type than— let me think, remember what the situation is. Oh, right. There are two classes of registers that are weaker than an atomic register. An atomic register is what you’d simply think of as reading and writing happening as if they were instantaneous. One is called safe and the other is called regular, and the paper “On Concurrent Reading and Writing”; it’s safe registers are automatically regular, not automatically atomic, and safe registers are trivial to build. They’re basically any way you might think of building a register would pretty much have to be out of multiple pieces would wind up having to be safe. “Safe” just means you get the right answer if you read it while nobody is writing it. So, I considered the problem of implementing-- well, first regular registers using safe registers, and then atomic registers using regular registers. While I was able to solve the first problem — not optimally, but at least solving it — and I believe that Gary Peterson later published a more efficient algorithm for doing that — and for going from regular registers to atomic registers I was only able to solve it for two processors, that is, a writer and a single reader. I proved the interesting result that you couldn’t do it unless both processors were writing into something. By “doing it” I meant with a bounded amount of storage. It’s easy to do it if you have unbounded counters — well, somewhat easy, but not in a bounded way — and so I was never able to figure out how to do it with multiple readers, and after I published my paper on it, there were a couple of papers published. One which claimed to have an algorithm that I have no idea why the authors believed that could possibly work, <laughs> and then there were a couple of later algorithms that were really complicated and I realized that the reason I never solved the problem is that the algorithms, the solutions, were sufficiently complicated that when things got that complicated I just would give up. I don’t know — sort of a combination of aesthetics and of not knowing if I had the mental power to deal with anything that complicated.

Levin: Was it your thought that this framework for different kinds of registers with varying degrees of complexity — or utility maybe I should say — was likely to lead to something else, or were you really focused on the problem that you had exposed in the bakery algorithm which was that you have these unbounded counters you have to deal with somehow?

Lamport: Well, I said, the problem with unbounded counters led me to think of what it means to build one kind of counter out of-- one register out of bigger ones, out of smaller ones, and then that led me into considering these three classes of registers, and the safe register is one that I said that it's-- I know that you can build them out of hardware, that's clear — and so that was the only register that I could say that physically I knew how to build, and again, I always regarded this as a physics problem.,

Levin: Exactly

Lamport: And actually when I started doing this stuff it was fairly early, I think around '75 or so, and I gave a couple of talks about it, but nobody seemed interested in it. So I put it away, and I thought I had solved the problem of atomic registers, and then sometime much later, I think-- yeah, it was around '84 — Jay Misra published a paper — I think it was Jay by himself, I don't think-- he published a lot of things with Mani Chandy, but I think that one was a paper by Jay himself — which started-- was heading in the direction of that work that I already did. So I decided to write it up and publish it, and when I got back to writing it up I suddenly realized that I couldn't imagine what kind of solution I had when I was talking about it in the '70s for the atomic register case because I probably was able to look at my slides and what I seemed to have been saying was utter nonsense, but anyway, so I sat down and came up with an algorithm, and fortunately Fred Schneider was the editor of that paper — as an editor he's incredible, was, I presume he doesn't edit anymore, <laughs> but he-- actually every paper he published he really read — and he went through the paper and he came to the proof correctness of the atomicity paper and he couldn't follow it. He said I don't understand something or other, and I said, oh, Fred, <inaudible> I'll go through it, got on the phone and went to explain it to him and it got to this point and I said, "By God, it didn't work." The proof was wrong, and in fact, the algorithm was wrong, and-- I've always been good at writing careful proofs, but it was before I had learned to write proofs the right way — hierarchically structured proofs — and so I realized the algorithm was incorrect so I went back to the drawing board and fortunately there was an easy fix to the algorithm and I wrote a much more careful proof, and that's what finally what got published. So I'm eternally grateful to Fred for saving me from the embarrassment of publishing an incorrect algorithm. Not completely — there was one that I did publish, but what happened is that it was a-- I gave a talk at an invited lecture at one of the early PODCs, and then it was decided I should transcribe-- I guess a recording was made. It was decided that I should transcribe the recording and turn it into a-- and publish it in the next PODC proceedings, which I did, and in one of the slides there was an algorithm that was nonsense, and I didn't pay attention to it when<laughs> I was-- I don't know how it got into my slide, but it would never have gotten into a paper if I had actually been writing it down, but I just reproduced the slide and it was not a major part of the talk, but it was just a silly mistake or silly piece of nonsense that I'd written the slide without thinking about it. But Fred saved me from what would have been the only I think really serious error in anything I've published, although it's an interesting error

that was actually in the footnote in the bakery algorithm. What I basically wrote, without using the terminology, but in the footnote said that basically a single bit was automatically atomic, and I discovered when I started thinking about atomic registers that that was wrong. It was nontrivial even to get a single atomic bit.

Levin: The obvious is sometimes not so obvious.

Lamport: Well, never believe that anything is obvious until you write a proof of it.

Levin: A good maxim. I want to pick up on a topic that you talked a little bit about earlier, but which became a major theme in your work on concurrency — really fault-tolerance — and it goes back at least to the SIFT work that we were talking about earlier on: Byzantine agreement — which of course wasn't called Byzantine agreement or Byzantine faults early on — became a topic that was pretty significant in your work and I'd like to--

Lamport: I suppose we should mention the story of why it did become the Byzantine Generals.

Levin: Yes, that would be a good thing to talk about.

Lamport: Edsger Dijkstra published very early on-- in one of his papers he posed a problem that's called the Dining Philosophers problem, and a lot of people have used that as an example of how to do this in some language or other — and received much more attention than I really thought it deserved. Edsger did some brilliant work and the Dining Philosophers was-- didn't seem to be terribly important to me, but it got a lot more attention than things that really deserved more attention, and I realize it's because it had a cute story about it involving philosophers eating bowls of spaghetti which required two forks and each philosopher just had one fork or something like that, and so I realized, I thought that the Byzantine-- that the problem was really important and that it would be good to have a catchy story to go with it, and Jim Gray had described--

Levin: The Byzantine agreement problem was very important.

Lamport: That was the Byzantine agreement problem.

Levin: I just wanted to clarify.

Lamport: A couple of years earlier I had heard from Jim Gray <coughs> a description of what he described as the Chinese Generals problem, and — it's basically an impossibility result that's quite

important — that I realized a lot later that Jim was probably the first one to state it and invented the Chinese Generals story to go with it. So I decided the story of a bunch of generals who had to reach agreement on something and they could only send messengers and stuff like that, and I originally called it the Albanian Generals, because at that time Albania was the most communist country in the world and it was a black hole and I figured nobody in Albania is never going object to that, and fortunately, Jack Goldberg, who was my boss at SRI said "you really should think of something else because, you know, there are Albanians in the world", and so I thought and suddenly Byzantine Generals and of course with the connotation of intrigue that was the perfect name. So that's how it got named, but you weren't originally talking about the name, you were talking about the problem. So what was your question?

Levin: Well, I think because this is such a fundamental thing, the transition from earlier work in which you assumed that things didn't fail to work in which failure had to be factored in — was in fact sort of almost fundamental — it seems to me an important transition and obviously it's important to deal with failure. It becomes ever more important in the modern world, so I'd like to hear how you got into that.

Lamport: That was obvious because I had had the "Time, Clocks" paper out with its algorithm for implementing an arbitrary state machine, but it was clear that when you had dealt with communication you had to worry about messages being lost and at the very least, so you had to worry about failures, and so that led me to say how can I extend the algorithm from the "Time, Clocks" paper, and make it fault-tolerant. And I guess, when I was through it didn't look much like the algorithm from the "Time, Clocks" paper, but as I said, it was an obvious next step. And I wrote this paper-- this forgotten paper in '77 about implementing an arbitrary state machine. And, in-- in the SIFT work, it didn't talk about-- they didn't think in-- talk in terms of state machines. They had already abstracted the problem to sort of-- it's a kernel of just reaching agreement. And then to build a state machine, you just have a series of numbered agreements that-- on what the state machine should do next. But, in the context of SIFT, there was-- Oh, SIFT was a synchronous system, so the system did something every 20 milliseconds or something. So there was a natural separation of an agreement protocol, done every 20 milliseconds and ordered by time. So when you get to an asynchronous systems, you don't have that every 20 milliseconds. And so, the-- you use what's effectively a counter. And that somebody to propose the-- the next <inaudible> action of the state machine you say, "Well, action 37 has been done so now this is-- we're now agreeing on action 38."

Levin: So that-- that becomes the sequencing mechanism in that Byzantine agreement is still the building block.

Lamport: Yeah.

Levin: It's just that you don't have a-- a real time clock, you have a logical time clock?

Lamport: Oh well, but the other thing that's changed, in the asynchronous world-- at the same time there was this sort of shift of thinking, at least in my thinking from the synchronous to the asynchronous world, there was also a shift in failure model, in that people were worried about computers crashing, not about computers behaving erratically and doing the wrong thing. And this can still be a problem in-- in practice, but it's a very-- it's considered to be sufficiently rare that it can either be ignored, or you assume that you have some mechanism outside the system to stop the system and restart it again, if that happens. I do remember once in — I think it was in the-- either the '70's or early '80's when the Arpanet-- I forget whether it was the Arpanet still, or the Internet — went down, because some computer didn't just crash, it started doing the wrong thing. And since the algorithms were presumably tolerant of-- or people worried about making them tolerant for crash failures but not of Byzantine failures-- malicious failures like-- that are now-- that are called Byzantine failures, or even just failures causing computers to-- to behave randomly. The Arpanet was not protected against that. And I don't know what they did to get it back up.

Levin: So this-- this leads us, I think to the-- to another project which became whole series of-- of works of yours around building-- building reliable concurrent systems and that was Paxos. You mentioned a little bit earlier that this was motivated by some work at SRC where you were trying to-- to show how something couldn't work, and instead you got an algorithm out of it. Can you talk a little bit more about that, and then tell us a little bit about how that work-- work came to the attention of the world outside SRC?

Lamport: Oh sure. First of all, the-- before that I had-- there was a book published by, what's the-- it was a database book. And-- but they had an algorithm supposed to deal with fault tolerance. And I looked at the algorithm, and I said that wasn't an algorithm to deal with fault tolerance, because they assumed that-- that there was a leader. And if the-- and if the system failed, well you just get another leader. But that's a piece of magic. How do you select a leader, and select a leader so that you never have two leaders at the same time, and what happens if you have-- that's just as difficult as the problem they claimed to solve. So, I just ignored that. And— which in retrospect, was a mistake, because, I think, if I had thought about the problem and say, not about just how-- how their algorithm doesn't work — but I thought about how do I make that algor-- how do I turn what they did into an algorithm that works, I think I would have gotten Paxos a couple of years earlier. But at any rate, as I said, the people at SRC were building the Echo file system and I thought — again this was-- this was a-- a case where we're just interested in crash failures. People believe that that's the way processors crashed almost all the time. And so, you didn't have to worry about Byzantine failures which was good because nobody knew at the time how to do Byzantine-- how to solve-- handle Byzantine failures, except in the synchronous world. And the-- it's not practical to build large synchronous systems these days. Or I don't know, maybe possible these days, but it certainly wasn't-- wasn't the way things were going in the-- in the world today, really in most of the computing because, to build synchronous systems, you have to have systems that will respond to a message in a known bounded length of time. And, the entire mechan-- way computer communication has evolved, there is no way to put a bound on how long it's going to take a processor to respond to a message. As you know, if you use the Net, and usually, the response comes back very quickly but it could take a minute to get a response. And so, we couldn't assume any kind of synchrony, so it was asynchronous system. And I thought that what they were trying to do in an asynchronous system was-- was impossible.

There's-- well, I guess maybe the reason I may have thought it-- let me try to get a possible reason why I thought it was impossible when-- an interesting back story to that. When I was interviewing at PARC back in '85, I gave a talk and I-- I must have described some of the Byzantine Generals work, and I made the statement that, in order to get fault tolerance, or to solve the problem that I was trying to solve, you needed to use real time because in the absence of-- of real time, there's no way to distinguish between a process that has failed or a process that is just acting slowly. And I knew, I was-- I was trying to slip something under the rug which-- which I do in lectures because there's a fixed amount of time and sometimes I have to ignore a thing-- embarrassing things. Not because, I want to hide them, but just because I want to get through the lecture, but any rate, Butler Lampson caught me on that. And he said, "Well, you-- it's true what you say that you can't distinguish between whether something was failed or just acting slowly, but that doesn't mean that you couldn't write an algorithm that worked without really knowing whether a process had failed or not, but just does the right thing if there aren't too many failures or whatever." And of course, I-- I couldn't answer it on the spot. I went back home later and thought about the problem, and I came up with some argument that it was impossible, and unfortunately that email has been lost, and I don't remember what it was, but I-- I realized that it wasn't rigorous-- certainly what I know, this writing wasn't rigorous enough, and it just wasn't very satisfying and I never carried it any further. But I-- I just had this feeling that there was some-- some result there. And, some number of years later, and I don't remember when it was-- when it was published but there was the famous Fischer-Lynch-Patterson result, known as the FLP result, which essentially proved that what I was saying was impossible, was in fact, impossible. And the-- it was a fantastic, beautiful paper, and they-- what they said, was much simpler and much more general than anything I was envisioning. So that was one of the-- one of the most, if not the most, important papers on distributed systems ever written. But any rate-- that experience may have made me thought that what the Echo people were trying to do was impossible. Although, I think it was that the FLP paper came later, but I'm not positive about that. At any rate, so I sat down to try to prove that it couldn't be done. And, instead of coming up with the proof, I came up with an algorithm. I mean, to sort of say, well, an algrothim to do this, has to do this so as that can't work because of something. Well, actually this could work because of that something, but you do something else and that following suddenly you say, "Whoops, there's an algorithm here." <laughter> And but it's interesting the relation with the FLP result. The FLP result says that in an asynchronous system you can't build something that guarantees that-- that consistency in-- in reaching consensus. That is where you're never going to get two different processes that'll disagree on what value is chosen. But it says, you can't guarantee that a value eventually will be chosen. And there's-- people have showed, I'm-- that, this is an explanation, post facto explanation, not something that's going through my mind at the time — but it's been shown that there are random solutions to the problem which don't guarantee but guarantee with-- with high probability that you'll-- that is, as time goes on, as the algorithm advances, the probability that you won't have reached a decision gets smaller and smaller. And so, what I would say that Paxos does, is it-- it's an algorithm that guarantees consistency. And it gets termination if you're lucky. And it makes-- it sort of gives engineering hints as to what you do, so that you have a good probability of getting-- of being lucky. And sort of reduces the problem to one of being lucky, which has engineering solutions. But even if you're not lucky, you're not going to lose consistency. And that's what the Paxos algorithm does. And that's what I came up-- up with. The problem with the paper was having been successful with the story with the Byzantine Generals. I decided to write a story for the Paxos paper, and it was about-- reason it's called the Paxos paper is it's-- the story line is, this is a paper's written by an archaeologist of this ancient

civilization of Paxos and they had this parliament and this is how they built the parliament. And I had a lot of fun with it — the paper is called The Part-time Parliament. And so I give examples of— as part of the story or how it would be used is— I would have the story— and gave the names of the characters Greek names, or more precisely, their actual names transliterated into Greek. And, I had Leo Guibas' help in-- in doing that. I had to add a few non-existent symbols to the Greek language to-- <laugher> to get some-- some phonemes that weren't in-- in Greek. But, and I thought that people would be able to know-- people who do any kind of mathematics should know enough Greek letters to be able to know, what an alpha and a beta looks like. And so, I'm reasonably close to-- if not figuring out the names, at least be able to keep-- to read them in their heads, so they'll be able to understand the-- the story. But apparently that was beyond the-- the capacity of most of the readers. And so, that part confused them. And-- oh, I submitted the paper to TOCS, I believe, Transactions on Computer Systems. And the referees came back and said "Well, this is interesting. Not a very important paper, but, you know, it would be worth publishing if you got rid of that-- all of that Greek stuff." And I was just annoyed about the lack of humor of the referees, and I just let it drop. Fortunately, and-- and almost nobody understood what the paper was about except for Butler, Butler Lampson. And he understood-- he understood its significance in that this gives you a way of implementing and in a-- in a fairly practical way, any system. Or, any-- a kernel of a system. The part that really required that-- that keeps the system working as a unison rather than-- keeps the processors from going off in their own way and getting chaos. And, you can use Paxos as the kernel of any distributed system. And he went around giving lectures about it. Also, in the-- at roughly the same time, Barbara Liskov, and a student of hers, Brian Oki, were working on a distributed file system, I think. And they had buried inside of their distributed file system something they called Timestamp Replication, which essentially was the same as the Paxos algorithm. And some point I heard Butler say that, "Well, Timestamp Replication is the same as Paxos." And I looked at the paper that they had published and I saw nothing in there that looked like Paxos. So I would-- I dutifully would-- when I wrote about Paxos, would quote-- would say, quote Lam-- Butler, as saying that, "Well this algorithm is also devised by Liskov and Oki," without really believing it. And then years later, Mike Burrows said, "Oh, don't read the papers, read Brian's thesis." And I looked in Brian's thesis and there, indeed, was very clearly, the same algorithm. But I don't feel guilty about it being called Paxos, rather than Timestamp Replication, because they never published a proof. And as far as I'm concerned, and algorithm without a proof is a conjecture. I think, Barbara has come to realize that. And I should mention that, she and her student, Miguel Castro — well, it was Miguel Castro's thesis — actually solved the problem of Byzantine agreement, asynchronous Byzantine agreement or Byzantine agreement in an asynchronous system. And there they had a very careful proof of correctness. And so, they invent-- essentially have what's "byzantized" version of Paxos.

Levin: So, you mentioned that the-- the editors and the reviewers were not particularly kind--

Lamport: Oh, right.

Levin: --to the paper. What happened to it?

Lamport: Well, one thing, I thought it was a really terrible paper. And some point, when a system was being built at SRC and, Ed Lee — he was there. And I think it was Ed Lee who-- who was told by somebody that, “Oh, you should read this paper about Paxos, because it’s what they need.” And he read it and he had no trouble with it. So, I feel not so much that-- less guilty about it. And willing to shift — to put more of the responsibility on the reviews and the readers. But at any rate, it was clearly an important paper and by the-- it was finally published around ’91, I think. And what happened is that — oh, names — Ken Birman. Ken Birman was the editor-- had become the editor of TOCS, and he was about to leave his editorship, and he said, “Gee, it’s about time that Paxos paper were published.” So he said, “I’ll publish it, but you just update it to talk about what’s been done in the meantime.” And I didn’t really feel like doing that. So, what I did is I got Keith Marzullo to-- to do that part for me. And, to carry on the story of being an archaeologist, it was that this manuscript was found in the back offices of-- of TOCS. And, the-- the author was away on an archaeological expedition and could not be reached. And so Keith wrote this-- I forget what it was called, some explanation or it was-- which was put in separate text as his contribution. So he did the annotations and it was published.

Levin: So I think if I-- if I got my chronology right — the work on Echo was in the late ‘80’s that inspired you to work on Paxos. And the first shot at sub-- at submitting the Paxos paper was in the early ‘90’s. And the paper didn’t actually appear until late ‘90s, is that right?

Lamport: Oh sorry, sorry. It’s the late ‘90’s that it-- that the paper appeared. We had-- we need to check the date on that. I don’t remember the-- the publication date of Paxos, but I remem-- I recall it being approximately 10 years after it was submitted.

Levin: Yes, that’s early ‘90s and late ‘90’s.

Lamport: Yeah.

Levin: But what stuck-- stuck in mind there. And so the-- the sort of core idea in-- in Paxos then turned out to have rather long legs, in terms of the work that you did-- that you did subsequently in applying the-- the idea again in different situations with different constraints that lead to different solutions. Can you talk about sort of that space of papers? You mentioned some of them earlier, at least in general. Disk Paxos, and Fast Paxos, and so on.

Lamport: Okay, the next one was, as I mentioned, Disk Paxos. And, when we were writing the paper, Eli sort of said, “Well, it’s really exactly the same as Paxos.” And I started writing the paper that tried to make them-- Disk Paxos look like just a variant of Paxos. And in fact, Butler Lampson, has written a paper called, “The ABCD’s of Paxos”, who claims that he has one germ of an algorithm from which he derives both ordinary Paxos, Disk Paxos, and Byzantine Paxos — the Liskov-Castro algorithm. But, I looked at the paper — I haven’t tried to look at in great detail — but I’m really skeptical, because my experience

with the Disk Paxos is that, you looked at a high level, it seemed that way, where you try to work out the details, it didn't work. And there was a-- just one, basic difference in the two algorithms. So they really are separate algorithms and there's no—"ur"-algorithm that-- that they could both be derived from. Since that time there was another thing that happened that algorithm called Cheap Paxos and that happened because a Microsoft engineer — I had moved to Microsoft by that time — needed an algorithm that-- well the-- for tolerate a single failure, you need-- with ordinary Paxos, you need three processors. But, most of the time two processors are enough. And the third processor, sort of-- you can sort of think of it as a backup, but in particular, if the-- it doesn't need to keep up with the other two processors until there is a failure. In which case, it has to take over from one of the two processors. So he went-- the, I believe it's Mike Massa, was the engineer, Microsoft engineer. He had the idea for using an asymmetric protocol that most-- that most of the time just used those two processors and then they would have some either-- some weak processor, or just a third processor use-- have some other computer that's busy doing other things, but can-- can come back in when needed. And, he had the-- a vague idea of how to do it and I came up with the actual-- the precise algorithm, wrote the proof, and that's how that paper came. Since then, I did some other work with Lidong Zhou and Dahlia Malkhi about— having to do with reconfiguration. One of the problems in fault tolerance is, say, for Paxos, you need three processors to tolerate one fault. Well, one processor dies, you need to replace it. Now, if it's really just three processors, then you bring in a new-- a new computer and just call it by the same name as the old one and it keeps going. But sometimes, you really want it to move things from-- to different computers. So you want to do some kind of, what's called, reconfiguration, where you change the processors that are actually running the system. And Paxos or in the state machine approach, it's a very simple way of doing that. You let part of the machine state be the set of processors that are choosing the commands. Well, of course you have a chicken and egg problem if you do it the naïve way, 'cause you have to know what the current state is to choose what the next command is. But how do you know what the current state is? And so, the-- so what was proposed in the original Paxos paper was, I said that the-- the Paxons did it by making the state, the time T, the-- that the processors would choose or the legislatures would choose the command at time T, be the one's according to the state at T-minus-3. Or command number T-minus-3. Well, three was a totally arbitrary number. I figured that everybody would know that-- realize it's a totally arbitrary number. But a lot of people kept saying, "What's three? Why did he choose three?" And something. At any rate, there were-- so at any rate, we did a few improvements, optimizations, or-- how to do reconfiguration. And Dahlia said that, "This method of putting it into the state isn't going to fly with engineers because they don't understand it." And so she-- we worked out a different procedure which was actually a variant of Cheap Paxos. And, but I don't think much of that work has had significant impact. I think, it's the basic Paxos algorithm that's the one that's still used.

Levin: So say-- say a little bit about the impact that it's had on-- obviously, it got-- it got used by people who were in your immediate vicinity, 'cause they learned about it quickly. But with the difficulty of the-- the word getting out through this-- this hard-to-read paper, and so on — how, how did Paxos become, shall we say, a standard-- a standard thing in building a large scale distributed systems?

Lamport: I really don't know what happened. I'm sure Butler had a-- a large effect in going around and telling people about it. I know, it's to the point where-- an Amazon engineer said to me, "There are two

ways to build a reliable distributed system. One, you can start with Paxos, or two, you can build a system that isn't really reliable." <laughter>

Levin: So they obviously believe-- or he did. Are there other companies that you know of that have-- have adopted it?

Lamport: Oh sure. Google, their-- I think they call it the Chubby Locks or something like that uses Paxos. I think there are three implementa-- I've been told of that there are at least three implementations of Paxos in Microsoft systems. It-- it has become-- well, it has become standard. There are other algorithms that are in use. I-- that I don't know about. There's an algorithm called Raft which is very simple that the—well, the creators of Raft claim that one of its advantages over Paxos is its simplicity. I take that with a grain of salt. I heard somebody recently give a lecture on-- on Raft, and it didn't seem too simple to me. But, I think that what's going on is they're measuring simplicity by, you show it to something, and people say, whether or not they say they understand it. And to me, simplicity means, you show it to people and they can prove it's correct-- then prove it's correct. And somebody else had told me is that Raft is basically optimization of Paxos. There's plenty of room for optimizations in Paxos. Paxos is a fun-- it's a fundamental algorithm and there are lots of-- of optimizations you can make to it. There's something else called Zookeeper, but I don't know what its relation to-- to Paxos is.

Levin: And at least some of this is in the open source world these days, right?

Lamport: Yeah, I'm sure there are open source implementations of Paxos since the patent, I believe, has expired.

Levin: We'll talk about patents a little bit later maybe.

END OF PART 1 0f THE INTERVIEW

PARt 2 OF THE INTERVIEW November 11, 2016

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...

Levin: <inaudible>

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

[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."

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 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?

<At this point the audio level dropped dramatically – perhaps Lamport accidently covered up his microphine?>

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.

<audio level returns to normal>

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