Welcome to MIT's Computer Science and Artificial Intelligence Lab's *Alliances* podcast series. My name is Steve Lewis. I'm the Assistant Director of Global Strategic Alliances for CSAIL at MIT. In this podcast series, I will interview principal researchers at CSAIL to discover what they're working on and how it will impact society.

Gerald Jay Sussman is a Professor of Electrical Engineering at the Massachusetts Institute of Technology. He received his PhD degree in mathematics from the institute in 1973 and has been involved in artificial intelligence research at MIT since 1964.

His research has centered on understanding the problem-solving strategies used by scientists and engineers with the goals of automating parts of the process and formalizing it to provide more effective methods of science and engineering education. Sussman has also worked in computer languages, in computer architecture, and VLSI design. He is an accomplished textbook author, including the Introductory Computer Science textbook used in MIT for 23 years. He has received the ACM's Karl Karlstrom Outstanding Educator Award and the Amar G. Bose Award for teaching.

Professor Sussman's contributions to artificial intelligence include problem-solving by debugging almost-right plans, propagation of constraints applied to electrical circuit analysis and synthesis, dependency-based explanation and dependency-based backtracking, and various language structures for expressing problem-solving strategies. Jerry, thank you very much for your time this afternoon. Why is software usually so brittle?

Well, there's a variety of kinds of software. Some of it's produced in the academic world and some of it's produced in the commercial world, and the reasons for being brittle are somewhat different in those two cases. Academic software is brittle because the people who are doing it have no incentive to polish it for production for use by others. It's to get a job done, specifically an experiment that's being worked on or a-- for later polishing by someone else that they don't know. And as a consequence, that doesn't give them much incentive to do that.

The commercial world has a completely different problem. Their problem is, its time to market is essential to living in a society where the network effect determines who makes the profit. So the fact is that the pressures in the commercial world to get something out very fast, whether or not it's any good, is very important. And then because you're the first to market that feature or something, and by golly, people get stuck on it, and then they'll keep using it even if it's crap.

So that's the, as far as I can see, why the main problem for brittleness. The difficulty is that anything that's brittle is going to be unreliable, it's going to have the problem that it will not be maintainable, and therefore, it'll have to be done over and over again. And that's exactly what I'm upset about.

You don't think there's any merit in let crowdsourcing QA-- get the product out there, have people kick the tires, find bugs for you--

That's fine, that's fine. That's-- I'm not worried about that at all. I'm worried about the fact that there's no incentive for people to do that much. But even so, if the product is put together poorly, the QA doesn't help, because you can't fix it. The best you can do is rewrite the whole thing, and that's bad.

OK. So do you think this is more of a design issue, architectural issue? How to go about designing better software?

I think part of it is incentive issues. I think that's the most important ones. But also, I think that there is a-- I think the engineers pretty much know what the right thing to do is in general. Real engineers. That's a very different question from-- I gotta be careful, because I think a lot of software engineering is not the kind of engineering I'm talking about.

If I were building electrical circuits, I know what I have to do. I have to make things fit tolerances, I have to make sure that they're manufacturable, things like that. Do we have that in software? No, because it's too easy to change, and flip and go very, very fast. You change it-- start all over again, maybe even easier often than try to fix some old thing.

I see. And could you describe what is additive programming?

Well, additive programming is an idea that many of us have had over the years, which is, you would like programs that have the property that they are flexible in the same way that-- in such-- a very specific way. In ways that, I think, can be adapted to new situations rather easily, and the way you do that is by adding stuff to it, not by changing the stuff that's already there. That's a very nice thing if it can be done. So it's very hard.

There are technologies that I've been working on over probably the 40 years of my being a faculty member here-- 45, I suppose-- that have been trying to improve that kind of thing. Mostly for myself. Because I like to have systems that last a long time. I'm lazy.

Can you give us an example?

Oh sure. An example of-- well, I'll tell you one that's not me. There's a very famous program called EMACS which is a text editor originally used for by programmers. It's still used by a very large number of programmers. It's over 40 years old. Probably 10,000 people have contributed to it. It has gone through major revisions, like adding Unicode to it, which is an enormous transformation, and it survived, and it works. And it's there and it works reliably independent of the fact that it's probably had 10,000 contributors.

Open source, I assume.

It's free software. It's not just open source. It's software that obeys the rules of free software. It's GPL. But that's an example of a thing that's lasted forever. It's real quality. It's tough as nails, it doesn't crash.

And was that-- with a program like that designed with this openness in mind, is additive--

Absolutely. It was-- the version-- the original version was done by Richard Stallman, who worked for me at the time. It's a very beautiful design. It's based on an interpretive language. So there's a-- and the extension language is basically interpretive, and as a consequence, it's easy to add things to it.

And that the underlying structure is almost like an operating system. That is, it protects its own memory, it does its entire memory management very clearly. And that's-- it's a very high-quality build, and that's why it survived so long.

Do you think it has anything to do with the simplicity of the actual application versus-- a text editor is not as complicated--

Oh, but even-- but come on, now. When you have-- that's true at the beginning. But the text editor has accumulated so much stuff that it's amazing. It knows every computer language. It knows how to format it, it knows how to-- it knows how to highlight the key words, it helps you with the indentation and things like that.

All of that works out beautifully. It knows about how to interact with the compilers and the debugging systems for things, and it really connect directly to the operating system in beautiful ways. It's not-- it's a huge program. It may be one of the largest programs in the free software world.

How about that. Can you give us another example?

Sure. Well, there are things that started out being in universities, like-- well, I think a lot of what-- I think Wolfram's Mathematica thing is an example. It's a program that's been added to for a long time. It started out originally from things that were-- at MIT we had Maxima, which was done by Joel Moses and Bill Martin. But that-- a lot of the ideas for the later things came from there. But the bottom line is that that's a thing that grows and expands.

We see that-- but we see that in other worlds, too. We see cities. A city is like that. It adapts to the changes. A city is not a program, it's a different thing entirely, but it has a property that as the world changes, the city adapts to fit it.

And that's-- so what you've noticed, for example, is amazingly enough, something like New York City has this wonderful subway system underneath it. That was probably put in at the beginning of the 20th century, mostly. It was very expensive, but boy, that made that city work. And I suppose Boston had an earlier one that didn't quite work as well because they didn't integrate it as deeply into the system.

Interesting analogy. What is Postel's law and how can it be applied?

OK. Well, this guy-- first of all, who is this guy? The guy was the guy who ran the internet, basically. He was the guy who assigned what's called the domain names until he died, basically, and it became part of-- it became an organization that did the work he did. He had a great idea about making things work well.

Which came originally, as far as I'm concerned, from digital electronics. The reason why digital electronics works well is-- and works very well is because say a particular piece like an inverter, which is a device-- an electrical device has the property that it can see-- 1's or 0's don't have to be perfect and they go in. They can be jiggling them, they could have noise. What comes out is perfect. So it has a range of inputs that it can accept that will then turn into good stuff.

Well, Postel's law is that. Basically says be generous in what you accept and be very precise and specific about what you put out. And good programs can be adapted that way, too. Why should it be the case that a program that takes a temperature as one of its inputs should care whether it's Fahrenheit, Celsius, Kelvins, anything like that?

It should be able to handle all that stuff. And that's very important in real programming, because what happens is then there's wiggle room. When you change something-- supposedly something's built on that. Well, you change something that doesn't have to change. It gives it a certain kind of stability and flexibility.

Interesting. In what ways can proofs of correctness be limiting?

Ooh. Well first of all, I like proofs. I'm an old mathematician. So I'm not trying to say that we shouldn't have proofs. It's not a question about the proofs of correctness that are limiting, it's the requirement that a person must prove things in order to make them work. It's the idea that every program, you want to make it as correct as possible, which is a problem.

Because this-- it produces, again, a kind of brittleness, and I'll explain. Supposing you have the requirement that every part of a program has to be completely approved in order to build the next point on top of it, well everybody is going to-- a good programmer is going to say-- or a good engineer says, I've got to do this, I gotta work out the simplest possible way. I have to use only the most restricted possible procedure to do this job so I can make an easy proof of it.

I can't give it that wiggle room, because that makes it more complicated in my proof. As a consequence, all the pieces are piled up on top of things that are exactly perfect. And if you put together a diamond, you can't modify it to make a difference-- to make a slightly different thing. You can't take a little piece off here and move it over. That doesn't work.

So you have to-- you end up with something that's very stiff, brittle, and works for one problem. You change the problem, it's over. Whereas if you allow people to use a little bit more flexibility and then they're not requiring them to prove things, but there are certain parts that you have to prove, of course, I will certainly tell you that the memory management system of an operating system better be perfect, otherwise it's going to produce junk.

What's going to happen is that it's going to crash in ways that nobody can debug. That's a garbage collector in the programming language. It has to be perfect. But that's a tiny program. It's only maybe two or three pages. That's it. So that I can understand. But I think the idea of things that say-- most things don't need to be proved anyway.

Supposing a Google search, one out of 100 cases produced a particularly wrong answer. You think anybody would ever notice? No. So it's not important. On the other hand, financial things have to be correct for law, legal reasons. But then proof may not be the way to solve that problem, that's why you have auditors. The auditors provide a second check on everything.

Does AI play any role in making software less brittle for me making more-- being more flexible?

Maybe. I don't know. That's a problem, because AI is a-- what is AI? AI is the part of computer science we don't know how to do. That's always what it is. Go back in history. In the 1960s, it was not even obvious the machine could play chess. There was a philosopher by Hubert Dreyfus who wrote a book that basically said you couldn't. Of course, a computer beat Hubert Dreyfus.

But that turned out to not be true. But then playing chess stopped being AI as something you knew how to do. People in the 1960s weren't so sure that you can make machines that could do symbolic algebra. Specifically symbolic integration of expressions. Symbolic integration of expressions used to be thought it was a very intelligence-intensive activity. Great mathematicians spent time doing that. And they compiled big books full of the symbolic integrals that they could find.

The fact is that we figured out how to do it. That was mostly done by Joel Moses, and eventually there was a Risch algorithm that did it perfectly, but Joel Moses made a big-- there was first Slagle, then Moses, and then the Risch algorithm And now it's no longer AI. You can go buy it from somebody. And AI is-- AI is always the part you don't know how to do.

So-- and of course, the big current change is that AI is now the part of making things that you can see pretty well. Guess what? It doesn't work very well. That's-- there's effort on that, of course, and that's good. So as something becomes-- as something becomes well understood, it stops being AI.

There's-- the history like this. Philosophy is that, too. And as something becomes clear, it becomes the science. Philosophy spins off things like physics and chemistry and things like that. That's very well-defined by Aristotle anyway in 300 BC or something like that.

So do you think AI has a PR problem, that people really just don't know what it means?

I don't think-- well.

Or what they think it means is--

I think that-- again, it becomes a buzzword right now. You can sell something because it's AI. Great. The important thing is whether or not what it actually does. And eventually they'll be very smart machines. Yes, I'm very pleased about that. I'd like machines to be smart. This machine that you have here is not very smart, you see. It may be conscious, that's a different question.

Is it?

Oh yeah. It knows-- it knows about its surroundings, it knows about-- it has senses, it feels what's going on. It can report upon things that don't work inside it. It can complain when it's getting pain. Come on now, it's conscious. It's just dumb. That people think it's conscious has something to do with intelligence, that's not true. You got plenty of conscious machines that are not too smart. Yeah.

Interesting. For everyday users, how will these improvements better their experience if we make software less brittle?

Oh, I think it's very clear. Stability. That's one of the things. How many times do you complain about the fact that somebody changed the operating system on you and all of a sudden your applications don't work? That sort of stuff. That happens like once every few weeks around here. How much-- in the case do you get some horrifying bug in some program you use all the time and you try to complain to somebody and they say, well gee, maybe you wait for the next version or something like that?

I mean, these are things that are not just annoying, they take up huge amounts of human effort. And I think that that's the kind of thing that I think will be better if there's a more smooth way to put together programs so that they can gently transform.

So why do bugs exist in software?

Bugs exist everywhere, not just in software.

Well let's talk about software.

No, I'm going to do it more generally. Bugs are-- bugs are necessary. They're a part of the fundamental problem-solving of an engineer as follows. You want to build some complicated system. You have to have a plan. The plan means that the systems may add various parts. You have to figure out how to put those parts together to make the whole. You have an approximate idea which you try, and it doesn't work, now you have to fix the interfaces. You have to make things-- you have to figure out how to make it work.

That's-- if it worked from the beginning, it wouldn't be-- you wouldn't be able to make something very complicated. This happens-- not just even in engineering, it happens in real poetry, darn it. A good poet is trying to manufacture an experience for the reader that changes-- that gives them-- and puts them in an emotional state.

Famous paper by Edgar Allen Poe. What's it called? "The Philosophy of Composition," you could look it up on online. Is about how he constructed *The Raven.* And he did it in exactly the way you're describing. He's saying that putting things together in order to make the appropriate emotional state. I pick parts and I put them together. I have to fix the interfaces, I have to smooth this place out and all that. That's exactly what an engineer is doing.

So I think that building anything complicated to have a particular purpose, there will be bugs and the bugs have to be fixed. But you can't avoid the bugs because they're part of the design process.

In brittle software, making software less brittle, do you think that will introduce more bugs?

It'll make it more bugs at the beginning and fewer in the end. What it means it'll make things more debuggable. You want to make you want to make the software understandable, you want to write it so that a person can read it. I wonder if the best things about a good piece of software is I can give it to you and you can read it. Guess what? You could learn something from it.

You could learn-- what's the software, it's how to do something. Well, wouldn't you like to know how to do something very complicated that you can't write down in English? Sure. So I'll give you a piece of code. If it's well-written, you should be able to read it.

Why is having a breadth of research interests and passions important to computer scientists?

Well, I think it's important to everybody, actually. But the--

I agree.

The critical reason is-- the critical reason for anybody who's going to build stuff that's complicated is to be inspired by the problem domains around them. If it weren't for-- if it weren't for physics, hardly any mathematics would be invented. If it weren't for-- if it weren't for things like molecular biology, it probably wouldn't be the case that anybody would ever care much about the mathematics of genetics.

So what's going on here is that there's a huge amount of pressure on people to do things because they know other things that are related and that they can solve a problem outside of this particular little thing they're working on. So I think, for example, I'm always-- I'm always inspired by biological systems which are extremely flexible.

Your genome is about a gigabyte. I don't know if you know that. It's 3 billion base pairs. Each base pair is one of a choice of one of four, so it's 2 bits. So it's about a gigabyte. Maybe if you add the methylation, you get to 2 gigabytes if you really-- if you really pressed it, but that can't be any more than that. That's not a very big piece of code.

On the other hand, it builds a-- from a single cell a very complex machine, the person, that runs for about 70 years with very little maintenance, and which is able to fend off attacks by other machines of the same sort that like to eat him. And better than that, it's very flexible. Change a little bit and also you get a rabbit rather than a person. Come on now, we haven't had a program yet. We haven't learned.

But be inspired by that, wouldn't it be nice to be able to code like that? Wow. I want to be able to make code that has that property that it runs for very long periods of time, produces complex machines at a very little-- it's quite dense, and yet it's flexible. That's the reason, inspiration.

It definitely gives you something to think about, although I wonder how trying to mimic a biological system or human--

I didn't say anything about mimic. I'm not trying to mimic. I said be inspired by. Quite different. Mimicry is actually a bad idea. You don't want to make an airplane that flap its wings like birds. It's very different. But being inspired by the fact that there are birds or bats gives you some insight into what it takes to make something that flies.

Good point. What are some of the things that people in industry could learn from academia?

Well, I think that's a very complicated question. There's a lot an academic can learn from industry, too. You gotta be very careful, this is not a one-way thing.

The most important things that industry could learn from academia is that putting more people on a project doesn't make the project run any faster. I suppose that's well-known anyway. That's the famous *Mythical Man-Month* book. But there's a bug which I've observed in industrial code which is that the normal way people try to insulate themselves from turnover of their employees is by having lots of redundant people. They hire more people than necessary and put them on a project.

And what they're doing is they're trying to insulate themselves from saying that there's-- they want to make sure that-- they think of the program as interchangeable parts, that you could lose one, you could put one in. That doesn't work. That's just not a viable strategy for any kind of creative activity.

On the other hand, there is a things that academics could learn from industry, which is, among other things, the idea of how to plan stuff that you get the timing. How do you put out-- how you say I'm going to do this first and then this and then this, and that gets done? So the kind of planning necessary to put out a product is quite different-- and also a lot of testing that's done to put out a product is quite a different in industry than it is in the academic world.

I don't have to make sure that you-- do a very strong test on something to use a program that I've written. I do have to have confidence in it for my job, but if I want other people to have confidence in what I'm doing, I better give-- I better put some effort into very extensive testing, and that's important.

What would you say your proudest accomplishment is in the world of academia?

Students. I am very pleased about the fact that I've had about 45 PhD students finish working for me. Doctoral theses. And they're my friends. And they're fun. And that's probably the most important thing I could point out. And almost all the-- almost all of the actual research work I've done has been motivated by teaching, I don't know if you know that.

Most people don't realize that-- yes, going back through-- you walk through the various things I've done, you'll find that a lot of it comes directly from an intention to make a better way to explain something to someone. How can I say something clearly more and more effectively and give someone something you can read that they can understand? That's pretty much what I'm mostly have done.

Now I can think about other things, too. If you want to say about specific academic accomplishments, I would say the ability to rapidly connect together, design, and building of hardware to high-level goals is something that got me a lot of places.

I did the-- I was studying some dynamical astronomy worrying about the motions of the planets and things like that. I got a bunch of people to help me whip together a special purpose machine for doing orbital mechanics called the Digital Orrery that basically solved a 300-year-old problem having to do with the stability of the solar system.

But what I needed was the ability to start from the problem at the level of the mathematical description of the problem, working all the way through code to do the kind of evolution that I wanted with extremely high precision, attacking things like the deep problems of numerical analysis in this case. Then going all the way down to the design of boards full of chips to do the job. That's what I feel very strong about.

Facinating. And where can people go to find out more about your research, your work?

I think my web page at CSAIL. It's fairly extensive, it's got lots and lots of pointers, too. Mostly my students' work, and a few papers I've written, but not many. I don't believe in lots of papers. If you look at-- how many papers in his lifetime the Richard Feynman ever write? Not many. But they were good. That's what I care about.

Excellent. Well, it's been a fascinating conversation. Thank you very much for your time, Jerry.

If you're interested in learning more about the CSAIL Alliance program and the latest research at CSAIL, please visit our website at cap.csail.mit.edu. And listen to our podcast series on Spotify, Apple Music, or wherever you listen to your podcasts. Tune in next month for a brand new edition of the CSAIL *Alliances* podcast and stay ahead of the curve.

[MUSIC PLAYING]