VIEWPOINTS RESEARCH INSTITUTE

Extracting Energy from the Turing Tarpit!
Dr. Alan Kay
Viewpoints Research

ACM Turing Centenary Celebration
San Francisco, CA
June 15, 2012

Ubiqus/Nation-Wide Reporting & Convention Coverage
22 Cortlandt Street, Suite 802 - New York, NY 10007
Phone: 212-227-7440 ♦ 800-221-7242 ♦ Fax: 212-227-7524

Extracting Energy from the Turing Tarpit!

[START RECORDING]

DR. ALAN KAY: This is actually in the nature of a pep talk, and it's an optimistic pep talk about software and it's based on the work of many people here. I should mention that many of my heroes are in this room, and not just the people who got the Turing Awards, but some of the colleagues I've worked with over the years are my heroes. I think we wind up making ecologies when we're actually trying to do things—this has sometimes been mentioned—and these ecologies are critical because they actually require a wide variety of different talents. I've always felt that Turing exemplified a wide variety of talents inside himself—just pretty darn amazing.

So this pep talk is about the Turing Tarpit and what we might do with it, and about it. I was a big fan of Al Perlis; he was a fantastic guy, he was the first Turing Award winner, and early on, he said beware of the Turing Tarpit where everything is possible but nothing is easy. And what's interesting, he was talking about programming languages, but in fact, this is true of the computer itself. That's the problem, it's got all these wonderful degrees of freedom, and we have to glorify in those, because that's what makes it the most interesting machine that anybody's ever invented. And of course the problem here is—oops—and fall in and perish.

Now, I did a little research on tar pits, and I cannot find anywhere that any humans were ever found in a tar pit that were not put there specifically as a ritual burial. So it seems like humans were a lot smarter than sloths—sometimes you never can tell. But in fact, it looks like humans were pretty good at avoiding the tar pit, but we created a different kind.

We realized, oh, there are all these hydrocarbons and fossil fuels, and we wind up with this. This and one of the slums in Rio de Janeiro are my two favorite pictures of what I think of as software today.

The problem is we can't see them; we can only see 30 or 40 lines of code at a time and there's this enormous, festering, whatever it is. And so this goes from "whoops" to a "yikes," and the tendency we have, especially in this country, is to beat everything to death with a club, and what happens is the sub-goals become the goals.

So at some point, just building refineries become the goals, where really what we want is energy. It happens that 67 percent of fossil fuels are burnt to make electricity. Well, if we want electricity, there are other ways of doing it.

For instance, photosynthesis is pretty nice. I used to be a biologist, and what happens is through a beautiful series of chemical reactions and some physical pathways, it's pretty darn efficient at getting electrons. And electrons are what we want.

So we get a "yay" on that, and there are a lot of organisms that do this, and the ones that don't feed on the ones that do, and it happens that the basal metabolism in all living cells is somewhat similar. Looking at it from the energetic standpoint, it's really about just transferring electrons around.

And so there are a lot of electrons there and in fact, recently it was discovered that many bacteria extend microtubules between each other not for the purposes of sex but for the purpose of exchanging electrons. Isn't that incredible? So they can get a kind of a feeding from each other; they share, so socialized electrons.

And because there are so many of these there, Adam Arkin, one of my favorite engineers, who is also a biologist, says hey, just stick the right things into the ground anywhere on the Earth, including the Sahara Desert, and you've got a fuel cell made out of living microbes that are just happen to be there by the billions and trillions, and so in fact out of pretty much any bucket of dirt that you get outside, you can run a 60-watt bulb just by having the right kinds of things that you stick in there. Doesn't require any additional power.

And this is kind of a metaphor for, I think, what we should be thinking about in software. We have to think about what it is that we actually want to do, and I don't think our main goal is writing millions of lines of code. What we want it for is for it to do something. So we've heard this in various ways today, but I like this idea of let's try science and math; let’s not beat it to death.

And idea No. 2 here is this distinction between tactics and strategies when we’re building things. And tactical stuff is usually pretty incremental and somewhat similar to where you were. So bricks lead to piles and stacks, and if you scale them up, you get pyramids and walls. And they have a certain simplicity to them; anybody can learn how to be a bricklayer. And so this has a lot going for it; it’s very transferable and so forth.

But it has some limitations, and so there’s a strategic direction we could go, which is before we build things out of the materials, let’s think of things we can make out of the materials that will be new materials.

So in architecture, let’s make some arches—it’s where the term came from. And these things are incredibly efficient. The Romans made such good cement—aluminum silicates plus compaction—that many of these structures built thousands of years ago still survive today. If you’ve been in the Pantheon, it looks like it was built yesterday instead of 2,000 years ago, and in fact it looks like it’s made out of granite instead of the best reinforced concrete anybody has ever made. By the way, the Frost Giants in the Wagner operas came out of the inability of the Goths and the Visigoths to tear down these aqueducts in the Dark Ages; they concluded only giants could have built them. But in fact it was just people like us with some technique.

And the same thing is true in computers. Our building block is a comparison; we could call it the Sheffer stroke if you want, and again, we can go tactically and calculators, we can add a strategic element to store a program, a la Jacquard blooms. And this notion of storage models of computing is recapitulated through many of our languages. And the predominant amount of programming today, despite some fringe areas, is actually done using this storage model in a wide variety of forms including so-called object-oriented languages. So this is kind of the general thing you find today. Most universities train towards these.

So, we’re talking about Turing. A really interesting idea is this brick idea’s: Yeah, you can program a computer to make something that’s like the computer you’re on, but you can also make it do something, a computer that is very different from the computer it's on. And the first one of these that I ever saw was Ivan Sutherland's Sketchpad and it was definitely a computer; it was programmable, it wasn't programmable like any other computer had ever been, and it was completely different than the enormous TX2 computer that it ran on.

And one way of thinking about it, and Ed Feigenbaum blazed the trail here, is that a lot of computing today is done in terms of hows, and as far back as 50 and 60 years ago, people were already starting about, can't we just program in terms of whats. Another one was John McCarthy, whose early paper on the advice taker, an intelligent terminal agent that you could give advice to, that could deal with things in common-sense terms—LISP was actually invented to be a programming language to make that. That was an early idea, and of course Marvin Minsky and Ed Feigenbaum and so forth have talked about intelligent systems as terminal agents.

I'm going to focus, not so much on AI here, but just this notion of what do we get if we can find ways to think in terms of whats. Marvin called this semantic information processing.

I'm going to focus on Sketchpad here, because it's not only Turing's 100th birthday; it happens to be the 50th birthday of Sketchpad this year. And I'm sure Ivan has forgotten this, 'cause he's famous of forgetting things like this. Because he refuses to talk about Sketchpad, I thought I would say a few words about it.

It's famous for being the invention of computer graphics; it's less known that it was really the first object-oriented system that had the abstraction technique of objects, masters and instances. You could make a master rivet and create instances from it.

And the big knockout on Sketchpad was the programming of it was not done in terms of anything like a storage model but was done by telling Sketchpad goals that you wanted it to achieve, and it had three problem-solvers in it that would achieve those goals.

And one of the famous examples here is this truss bridge, and it happens that there’s no movie available of this truss bridge. And the reason is that TX2 took a while to settle the constraints on this bridge, I believe.

But because it’s the 50th anniversary of Sketchpad, we’ve recreated the Sketchpad display. And this twinkling, Ivan will recognize, because it actually plotted points and they had to be randomized throughout core, or you’d go crazy with vertigo on it.

So with Sketchpad you could add a truss in here; you could turn on the constraints, one of which is gravity, and Sketchpad would compute all of the beams there, and we can see what effect this one has of adding it in here.

So this is 1962, and a wait, and another wait. And this twinkling and stuff caused a series of better displays to be made, but it’s interesting that in many ways the semantics that are expressed here haven’t been improved much on, although, we come into modern times, we can make it look prettier.

And we have this model, is due to Yoshiki Ohshima and Bert Freudenberg, who made this. Now in the modern version, of course, words have weight, so I’ll say, Ivan, thanks for inventing this, and we’re doing the constraints a little bit differently here.

And so we put them inside this weight here. Here’s a couple of them. So here’s the spring constraint, so the beams are all springs here; this is gravity, and this is the pin constraint, so if I turn gravity off, the springs will pull it. I love the idea, just the idea that a physical object is actually a process.

Now it’s gone to zero; I’ll turn gravity back on. Let’s turn off the pin constraint. Now, the pin constraint is what is holding all of these girders together at the joints here. So just pull it off.

Okay, but we have all these useful objects down there. And we know these objects, what we’re seeing there are just costumes on these objects, so let’s repurpose them. We’ll give this system a different set of constraints. I’ll use the little balloon here to raise them up.

Okay, this is not such big news, because we do our layout constraints. So this is a user interface of the system I'm actually giving this talk in terms of.

Steve Jobs might like the steel look, but I think, being a little more colorful, we can try a blue look here. That looks a little bit too much like a company we're all familiar with, so, this system is sort of a Frankenstein monster, so I'll go with a Halloween theme here. And I've got, this is kind of like a presentation system except I can program in it, so what I'm going to do here is just go to the slide presentation, but in fact, I can still interact with it because this is just a view; there are no modes in this system.

And this brings us to what I think of as the third Turing machine. We talked about the first two; third Turing machine was the machines he built out of biology. And this is, in fact, the first Turing paper I wrote - that I read - when I was a biologist, was the morphogenesis paper. And even though I was a math major also at that time, I'll confess I did not understand the mathematics in it. It was deep. And actually he warns in the beginning of that paper what parts to read unless you're really into mathematics.

But one of the ideas here is he was he was trying to show that relatively simple systems of gradients interacting with each other can give rise to an enormous number of morphological structures. And these are some from his original paper, and these are computer generation of some of his models more recently.

And you can do other things with these. So gradients, we can do something that's kind of like particles in fields—here we've got little ants following the gradient and they'll all eventually migrate their way up into that upper left-hand corner and it looks like they're somewhat coordinated but in fact, they're all just following their own thing. And even though we could build our text-handling stuff in this form, it suggests, well, why not make the letters ants, and so if we tell these guys to interact with each other—and by the way, these three rules here are the rules for this text editor here, so you can sort of see what that looks like.

And another little wrinkle in here—that was done by Ted Kale—another wrinkle here is done by Aran Lunzer, and it’s that nowadays we have the computing power so that we don’t have to compute just one version of something, but we can compute multiple versions. So for instance here are three suggestions, three views of this paragraph computed all at once in different worlds, and we see the outline superposed over it, and we can sort of see how this works. And this underlines that when you’re doing what-type programming, you really want to have a user interface that is giving you a sense of what the possibilities are going on, because the system is doing more for you than when you’re doing sequential programming.

And of course, there’s the question of doing the graphics here, and so Dan Amelang did the graphics for this system and assembled away, using mathematical relationships first. Here’s a formula for the involvement of a polygon with a pixel, and then he made a mathematical language that allows us to express that in a form that runs. So you can think of this as kind of a specification that actually runs.

In light of the previous discussions, I’d say my prejudices are to make our specification language something that we can debug and then ship. Because the problem is you’re always involved in not just programming something, but you also have to do all the reasons for the programming, and those can have bugs also. But I kind of like this way. But I happen to agree with Butler that sometimes making things clean is difficult; but on the other hand, think of how wonderful it is when you can make them clean. So that does things like that, and then we have a zillion compositing rules, but there they all are, just 95 lines of code.

We have pen stroking, and like this, and we have gradients and texturing and stuff, and the best way to show that is to go back here and I’ll just pick something in user interface, because the whole system is live. I’ll say okay, gradient fill, and I’ll just re-customize my little thing here, because that’s the live thing that goes along with that. Okay, and then back up, and then bitmap filters and everything.

So in this form, all the 2D graphics that we’re familiar with on a personal computer is 435 lines of code. Yeah, you have to learn a language, it’s a little bit like APL meets Christopher Strachey, but in fact, the compactness of it, and the understandability of it, is fantastic.

And in order to make languages like that fast, we made a metalanguage. Alex Warth did this, and it converts expressions into object relationships that you can do things like compile with or interpret, or you can do something really interesting and illuminating with it. As Brett Vickers, a young, fantastic user-interface designer, Bret would say, eh, that’s not really interaction. You need to continuously experience the meaning of your meetings, not just hit buttons and have them happen.

And this is possible in this graphics language because it’s actually a dataflow system as well as a very higher-level language. Whoops, sorry. So here’s an example of the letter D, and here’s what Bret has done. He has automatically gone through that code that I showed you, and he has built this visualization over here automatically from it.

So here’s a Bezier curve, and I can edit this letter D here, and you notice, everything else is reacting. Here are the six Beziers, that becomes the input of the rasterizer, that produces 49 span coverages, and I can come down here and I can ask, where did this pixel come from; I can open these up, and what it does is it takes me on a tour automatically through the code forwards and backwards and shows me what’s going on. And of course the next step after this would be, instead of writing this stuff over on the right first, you might want to start off with your examples first and develop the stuff on the right.

Okay. And so an architecture for this is somewhat similar to things that have been talked about for a while. This is Alan Borning and Hassam Samini [phonetic], who have been working on this. And—basic idea is, whenever you can stay up at the what level, the constraint level, the meaning level, the relational level, things are generally good.

And you’d like to have a language or two, not too many of them, that really allows you to express these, not just for special cases. I mean, one of the points of this is, it’s time to start thinking of doing what-programming for systems programming—making operating systems and basic elements out of it, where the system is in charge of doing the hows, the solvers, the optimizations, the pragmatics. And there's zillions of these.

We want to work up here, and this is kind of Sketchpad-like. Sketchpad had three of these; we actually need a lot of them. And in between we need something that is not just a kind of a chooser but also a kind of an expert system that knows about this idea, that can make judicious choices and maybe even parallel choices to tie this together.

So when I look out on the world here and somebody asks me about the Turing Tarpit, I think it's a Turing opportunity, and if there's a tar pit around, I think it's inside of our own heads. Thank you very much.

[END RECORDING]