Getting Computers to Understand

BUTLER LAMPSON

Microsoft Research, Redmond, Washington

There have been three broad waves of applications for computers:

Category	*Starting*	*Examples*
Simulation	1960	nuclear weapons, payroll, games, VR
Communication (and storage)	1985	e-mail, airline tickets, books, movies
Embodiment	2010	vision, speech, robots, smart dust

I think of storage as communication with the past, and by "embodiment" I mean somewhat unconstrained interactions with the physical world. The dates are when the wave began to be important; it's interesting that they are about 25 years apart. Currently, the communication wave is in full flood, and the first signs of embodiment are starting to appear. Of course, old waves don't recede: simulation continues to be an important class of applications.

New applications are enabled to some extent by new knowledge, but mainly by Moore's law, that is, by improvement in the cost/performance of computing hardware: processing, RAM, disk, and communications. This has been running at about a factor of 100 per decade. Sometimes, as with speech recognition or web search engines, the cheaper cycles or bytes are applied directly. Often, however, spending more hardware resources makes it possible to spend much less programming effort, as with applications that use web browsers or database systems as components.

I find it much harder to predict what will happen in computing during the next 20 years than it was in 1972, when I thought it was fairly obvious [Lampson 1972, 1988]. Perhaps this is because I'm getting old, but perhaps it's because embodiment is much more open-ended than simulation or communication. In addition, dealing with the physical world depends on many things outside the domain of computing, such as mechanical effectors, neural connections, or radio. And it requires dealing with uncertainty, which is still poorly understood.

Nonetheless, I have chosen two problems whose solution will make a big difference both to computer science and to the larger world of which computing is now such an important part. One is an example of embodiment: cars that don't kill people. The other is a technology that applies across the board: writing programs automatically from specifications. The second obviously has many applications, but the first demands major advances in computing that will also have many other applications. They share a common theme: getting the computer to understand something; in one case roads, in the other specifications.

1. *No Road Traffic Deaths*

The problem is to reduce highway traffic deaths to zero, without reducing the convenience of today's road transport system (well, perhaps not exactly to zero,

Journal of the ACM, Vol. 50, No. 1, January 2003, pp. 70–72.

since nothing in the macroscopic world is perfect, but a problem should be easy to state and remember).

This can't be done with better roads, because there are too many roads, and most of them often have dogs, children, or other uncontrollable objects on them. It can't be done by changing cars into trains, because trains can't go to most of the places that cars go. The only way to solve this problem is an automatic driver: cars must drive themselves, at least in emergencies.

This is an almost pure computer science problem, since the sensors (cameras, radar, laser rangefinders, GPS, etc.) are already much better than a person's, and the effectors (throttle, brakes, steering) are already computer-controlled or can easily be made so. We don't have to build hands or learn to walk.

The main challenges are:

—Real-time vision. Other sensors can help, but only vision can deal with all the complexities of real roads.

—Good models of roads (including their borders, which are much more complex than the roads themselves), of vehicles (both appearance and dynamics), and of foreign objects that can intrude on roads. These are things that a driver learns over many years.

—Dealing with uncertainty about sensor inputs, vehicle performance, and how the environment might change in the near future (What if that pedestrian steps into the street? What if the car in front brakes suddenly?).

—Dependability. The automatic driver must do the right thing (or *some* right thing) whenever there's a risk of death. It must not crash, unless the car has stopped safely, or at least slowed down enough that the driver can be trusted to control it. The option to stop or go quite slowly is the one thing that makes an automatic driver easier than an airplane autopilot; everything else is harder.

What about deployment? Of course, it will take a long time to equip every vehicle, but one might expect the benefit to be at least linear: each car that's equipped won't contribute its current 2×10^{-4} deaths per year. There are liability issues, but today we equip every car with at least two devices that explode in your face, so surely we can figure out how to have automatic driving.

Of course, automatic driving will have many other benefits. When the system is trustworthy enough, drivers can stop paying attention to the road and do something else. When most cars are equipped, they can talk to each other, coordinate their behavior, and use the roads much more efficiently.

2. *Automatic Programming*

The problem is to write a program from its specification as well as a team of programmers could do it [Gray 2003]; this is automatic programming. Once that phrase meant compiling, but a compiler bridges only a small part of the gap between the programmer's intent and a running program.

Automatic programming is not the same thing as "end-user programming," nor does it require applying common sense; these are both good problems, but they are not what I have in mind. I assume that professionals are writing the specification [Jackson 1995], and I assume a precise specification, perhaps written as a high-level state machine [Lamport 1989].

As stated, the problem is too hard (unless the team of programmers isn't very good, or doesn't have much time). How to scale it back? Perhaps the programmer can provide hints; these might take the form of a "high-level" implementation, as in SETL [Schwartz and Dewar 1986]. Perhaps, the domain can be limited; domain-specific knowledge can make the problem much easier. Perhaps performance can be sacrificed; thanks to Moore's law, it's usually acceptable to give up a factor of 10.

This is not a new problem; people have been working on it for forty years. There has been some limited progress:

—In some domains, declarative programming works. Spreadsheets and SQL queries are successes: the spec is close to the program. Programming by example is useful in text editors and spreadsheets. HTML markup is somewhat successful. Unfortunately, these solutions have sharp edges: spreadsheet macros, SQL updates, and precise control of layout in HTML are very ugly.

—Transaction processing is very successful: it turns a collection of independent simple sequential programs into a concurrent, fault-tolerant, load-balanced program without any extra work.

—Big components make a big difference. It's usually much easier to build a program on top of a relational database, an operating system, and a web browser than to build it from scratch. It may consume 100 or 1000 times the hardware resources of a custom-built program, but this is usually a bargain [McIlroy 1986]. Writing big components is very expensive, but it pays off big.

It's unclear how much these approaches can be extended. Certainly, they lack the generality that we would like.

Modularity is poorly understood, but surely important. There are ways to write a spec as the conjunction of partial specs. Combining implementations of the partial specs is largely unexplored.

REFERENCES

GRAY, J. 2003. What next? A dozen information-technology research goals. *J. ACM 50*, 1 (Jan.), 41–57.

JACKSON, M. 1995. *Software Requirements and Specifications*. Addison-Wesley, Reading Mass.

LAMPORT, L. 1989. A simple approach to specifying concurrent systems. *Commun. ACM 32*, 1 (Feb.), 32–45.

LAMPSON, B. W. 1972. Remarks on the nature of programming. Guest editorial. *Softw. Pract. Exper. 2*, 3 (July), 195–196.

LAMPSON, B. W. 1988. Personal distributed computing: The Alto and Ethernet software. In *A History of Personal Workstations*, A. Goldberg, Ed. Addison-Wesley, Reading, Mass., 293–335.

MCILROY, D. 1986. Programming pearls: A literate program (review). *Commun. ACM*. *29*, 6 (June), 480–481.

SCHWARTZ, J., AND DEWAR, R. 1986. *Programming with Sets—An Introduction to SETL*. Springer-Verlag, New York.