Title: In Search Of The Software Transistor

Authors: David Boskey and Tim Chase

Address: Corporate Computer Systems, Inc.
33 West Main Street
Holmdel, New Jersey 07733
U.S.A.

Telephone: (201)946-3800

Telex: 642672 CCSHOLM

> "In the beginning the computer was invented to solve the problem. What seems to have happened is that the computer has become the problem. So now the question is, what can we invent to ..."
>
> - Robert M. Baer
> The Digital Villain

In Search Of The Software Transistor

In this paper we will take a brief look at attempts to solve the problems which have been associated with software development. The reason we call it the search for "The Software Transistor" is that it was the development of the transistor which catapulted computer hardware into the advanced position which it holds today. In order for software to join those same lofty ranks, someone must develop the software equivalent of the transistor. We will, unfortunately, conclude that although some work is promising, there is still a long way left to go.

About Predicting

The theme of this conference is "Migration to 2001." With such a theme, it would appear appropriate to take a chance and make some predictions about what will be happening to software development at the turn of the century. This is a dangerous game, especially when the predictions are made about a year which will (hopefully) be reached by the authors. If we were to predict for the year 3000 our reputations for soothsayers would remain unsullied for the remainder of our lives. Undaunted, we will attempt to

sketch a brief picture of what we think is around the immediate corner for software development in general and programming languages in particular.

By their vary nature, programmers tend to be optimistic creatures. This was noted by Frederick Brooks in his, by now classic, book The Mythical Man-month. Brooks explains programmer optimism by saying that perhaps there is a natural selection process by which the frustrations of the job drive away all but the most optimistic. Whatever the reason, the trade is populated with optimists who survive mentally by believing that the project is really 95% finished or that this bug is the last one in the system. Predictions by optimists (especially those trying to get research grants) are bound to be tainted.

Marvin Minsky, a popular M.I.T researcher, in Artificial Intelligence was quoted in the November 20th, 1970 issue of Life Magazine (one of the US' better technical journals) as saying:

> "In from three to eight years we will have a machine with the general intelligence of an average human being. I mean a machine that will be able to read Shakespeare, grease a car, play office politics, tell a joke, have a fight. At that point the machine will begin to educate itself with fantastic speed. In a few months it will be at genius level and a few months after that its powers will be incalculable."

Poor Marvin, he committed the double error of being an optimistic programmer (a-hem, researcher) and predicting within his own life span. The point of this all is that programmers are often the ones who are making predictions about programming and computer science. This usually means that things are predicted to be much rosier than they really are.

What we will offer here is a slightly pessimistic prediction of the near future, but since we, ourselves, are programmers, the prediction will actually be somewhat optimistic. We hope that the two forces will cancel out and the result will be realistic.

History teaches...

Before looking into the future, it is often helpful to look into the past if only to discover that looking into the past is not all that helpful. Fortunately, for computer historians, computer science is quite young. We don't have to find fossilized printouts in order to get insight into the dark ages of data processing. Most people refer to "generations" of computer hardware. Although this is often just a marketing technique (any given vendor is always working on the "next generation") it is useful to contemplate the generations of computer hardware:

1. Electromechanical/vacuum tube computers. These were the first. They were large, unreliable, slow and often doubled as space heaters.

2. Transistorized computers. IBM's 7090 was one of the first, and some wistfully think one of the best transistorized computers.

3. Integrated circuit computers. Smaller parts made for logically larger computers.

4. VLSI (Very Large Scale Integrated) computers. More (less) of the same.

5. Computers from Japan.

The fifth generation computers haven't been born yet regardless of what vendors are saying. Most American Universities writing grant proposals feel as if the Japanese are on the brink of the fifth generation and that the US will lose its dominance in computer science unless more money is spent for research.

As luck would have it, there also appears to be 5 generations of computer software. This is especially obvious to all those folks selling forth generation languages. There is little connection between the generations of hardware and the generations of software other than faster computers can do more computing. It appears to be a fact that advances in software always require more computing.

As we see it, the five language generations are:

1. Ones and zeroes. Really the old days. This is where Grace Hopper got her start.

2. Assembly language. This includes macro languages, linkers and the like. It is amazing how many programmers still feel that there is something noble about assembly language.

3. So called high level languages. These include FORTRAN, BASIC, C, PASCAL, PL/1, LISP, COBOL and your favorite.

4. Programming environments. These are integrated facilities which combine languages, data bases, screen facilities which attempt to enable programmers to develop prototype and final applications quickly.

5. What ever Japan, Inc. picks for the fifth generation. More seriously, the fifth generation appears to be expert systems with a side order of nonprocedural programming. This is different than programming languages in the classical sense, as we shall see.

By looking at this brief history and by observing where we stand right now we may conclude some interesting things. The single most interesting conclusion we can make is that that hardware is far and away outpacing software in terms of progress. In the world of hardware, significant advances have been made just about every 10 years. These advances have led us from computers which filled rooms to computers which fill thimbles yet perform faster, cheaper, better, etc. The important thing to note is that there have been orders of magnitude improvements made in hardware development which come at regular intervals and are related to improvements in basic technology.

Software, unfortunately, is another story. If you include FORTRAN in the third generation of software language development then you find that we entered that generation on November 10th, 1954! On that date a document titled PRELIMINARY REPORT, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN was published by the Programming Research Group, Applied Science Division of IBM. This is amazing because the first computers had only come into being around 1948. This means that about six to eight years after the first generation of computers we were al-

ready into what we now consider the third generation of programming languages! Couple this with the fact that we think we are currently in the fourth generation and you have the basis for a depressing hint of what is to come.

Granted, FORTRAN does not embody all that is true and beautiful in current modern programming languages. The point we are making here is not that language development stopped in 1954, but rather that the changes which have come to programming have been small and have not even come close to having the impact on throughput that corresponding changes in hardware have had. We realize that some readers will respond violently to these charges; that there is a favorite feature of a favorite language which is being maligned here. To this we ask that you stop and consider the difference between a computer constructed from relays and a Motorola 68000. No programming language improvement comes anywhere near that level of change.

Why is there such a difference between hardware and software?

This is an important question. In order to answer it we must first begin to insult hardware developers. If you look at the changes in hardware development you notice one significant thing. The software model of computers has not changed much since the Beginning Of Time. By software model we mean how the "inside" of the computer is organized; the part the programmer sees. Again, we expect that there are some who will argue, but when you get right down to it computers have remained much the same since the beginning. What has changed with the computer generations is the technical implementation. Take the venerable IBM 370 as an example. It would be possible to implement a 370 in vacuum tubes. Clearly you might need Niagara Falls to cool it, and the G.N.P of a medium sized Latin American country to pay for it, but it could be done. Likewise, a 370 could be built using transistors and other discrete components. Finally, a 370 could be built from VLSI parts. In fact, it probably would only take one VLSI part. What we would see across the different implementations would be a vast range of performance with the vacuum tube 370 hopefully at the low end of the scale and the VLSI at the high end.

These so-called "technology remaps" have been used by computer vendors throughout the years to offer faster computers which still run the same software. The important point to remember, then, is that the "stuff" that computers are made

from has been changing but the design has remained steadfastly the same. When a new computer is announced, we all ask the same questions (how many registers, how many CPU's, etc). We are never surprised with the answers because the architecture is always pretty much as we expected. (It is interesting to speculate how well a really different computer would sell. Imagine you get the first look at a new computer design and find it resembles a fish tank filled with a rose colored jelly with wires sticking out from it and no one you have working for you has the slightest idea how to get accounts receivable running on it. How many would you buy? With economics as the master, perhaps we are getting exactly what we are asking for.)

So, hardware has the benefit of physics behind it. The hardware boys are innovative, sure, but they don't have to find vastly different organizational approaches to improve their product. A pipe line here, a parallel processor there and a heavy dose of solid state physics accounts for the orders of magnitude in hardware improvements.

Now, how about software? Well, software is a tough one. This is because programming is very much akin to thinking. Programming is problem solving. In a very real sense, programming is us. The difficulty is that it is hard to do a technology remap of our own brains. The implementation of the programming "machine" has remained constant over the last 40 years. It still remains "liveware." The problems associated with programming significant programs are problems which have faced mankind for ages. They are human organizational problems. How do you organize people so that they are all working toward a common goal? This is especially difficult if the goal is getting a computer to do something.

What is programming and why is it so hard?

One of the problems facing program developers is that programming is difficult, yet the popular concept of computers (from numerous Charlie Chaplin ads) is that they are easy to use. It may be true that computers are easy to use, but it is also true that they are difficult to program. This difficulty stems from the fact that the physical act of programming represents only a small part of getting a program out of a customer's head and into a computer.

Programming is much more than writing FORTRAN statements. A large portion of any job is spent in planning what the

program will do. Frederick Brooks says that at least one third of a project is spent in planning and only about one sixth is spent in actually writing code. Our own experience indicates that this is quite true. Further, as planning progresses the ultimate customer is often lost by the result.

The software developer wants to develop functional requirements which are detailed so that he knows exactly what is going to be built. The finished documents are often beyond the understanding of customers who are forced to sign off on them in order to begin development. Time allocated to testing is often used up by development which results from customers finally getting to try the system. The relationship between developer and consumer is often ruined by mismatched anticipation levels. Even with lengthy requirements documents, the customer often does not get what he wants.

In short, software development is a dirty difficult business. Regardless what the data sheets say, it is hard to write good programs which meet the customer's needs and anticipations. Our conclusions for the current state of computer science is that things have not changed all that much since the early days of programming. The big changes have come from the hardware side of the house -- no one has, as of yet, discovered the software transistor.

What does 2001 hold in store?

Hold on. This is where we start predicting. Software development has not changed significantly since the beginning. We don't see big changes in the near future. What we do predict is that programming computers will not get easier -- using computers for some, however, will get much easier.

If things continue as they are now, we see a sort of class structure developing. In H. G. Wells The Time Machine the world is peopled with two classes: the Eloi and the Morlocks. The Eloi are forever young and beautiful. They live lives of complete leisure while the Morlocks toil beneath the ground tending the machines which make the world work so ideally for the Eloi. Of course, in the end, the hero discovers that the Eloi are actually raised like cattle for the Morlocks to eat.

Except for the culinary twist, we see much the same for computers. There will be the Eloi who work with increasingly

sophisticated packages designed to enable them to use the computer without a great deal of effort. One of the technologies which will make this possible will most likely be what we now term "expert systems". Expert systems are a form of "declarative" or "nonprocedural" programming brought to you by the folks in the artificial intelligence labs. (Remember Marvin Minsky?)

The basic goal of nonprocedural programming is simple: tell the computer facts about the problem, toss in a few rules relevant to the solution and the computer does the rest. The Japanese in their Fifth-Generation project have (according to some reports) selected a programming language called PROLOG as the base for nonprocedural computing.

Nonprocedural programming is a good technique but it is not without problems. Consider the language PROLOG. Most would agree that PROLOG is a nonprocedural language and for small programs it does, in fact, appear to do just what is asked for. PROLOG allows the programmer to enter facts and rules and then ask questions about the data PROLOG "understands." The PROLOG system searches the facts and rules to derive an answer to the programmer's question. For small problems PROLOG does not need any procedural input from the programmer. However, for interestingly large programs, PROLOG grinds to a crawl. This is not too surprising because declarative languages spend most of their time searching the solution spaces defined by the facts and rules. The only way in which they may be speeded up, short of faster hardware, is to introduce (you guessed it) procedural programming to encode heuristics in order to trim the search space down to size.

In fact, we really have our doubts about the whole concept of nonprocedural programming. As Jean Sammet pointed out way back in 1969 in her book Programming languages: History and Fundamentals, the concept of nonproceduralness is really a very relative term which changes with the state of the programming art. To an assembly language programmer a statement such as

```
X = A + B * C
```

is nonprocedural. After all, we did not tell the compiler how to calculate the expression, only that we wanted to calculate it and where we wanted the results to end up. If you really understand the inner workings of a language like PROLOG (and you'd better if you're going to write any in-

dustrial-strength applications) then it becomes procedural. But, of course, it is not a very good procedural language.

If the Eloi use the expert systems who is going to build them? The Morlocks are the builders and they are faced with a double whammey. First, they must code the basic core of the expert system. To mystify the art, the basic core program is often called the "inference engine." The bad news is that the inference engines are "old fashioned" procedural programs with all of their associated problems (hard to write). Worse than that, expert systems introduce a new kind of programming called Knowledge Engineering (KE for short). If you think classic programmers have a bad time of it, wait until you hear what KE's do for a living.

It appears that expert systems are well suited for "consultation" programs. This is where the Eloi user sits down and chats with the computer to get some advice on what to do in a given situation. The example everyone sites is always the MYCIN program developed at Stanford University in the 1970's. Until MYCIN, most expert systems spent their days trying to beat humans at chess or tic-tac-toe. MYCIN was the first serious expert system. Its job was to act as a consultant giving advice on the diagnosis and treatment of bacterial blood infections (we're not talking pawn to king four here). Now, you might ask how did MYCIN get its smarts about blood? The answer lies in the KE. The knowledge engineer's job is to sit down with experts, to pick their brains and then to encode the expert's problem solving techniques into a data structure. The resulting "knowledge base" is the brains behind the expert system. If writing good programs is kind of hard, then knowledge engineering is down right difficult! For certain it is not something that the Eloi are going to be able to do on their days off.

As knowledge bases grow so does the potential complexity of the computer's responses. It is currently difficult to fully test and debug conventional computer programs. In the future it will be even harder to debug expert systems. In their most gross form expert systems are collections of facts and rules. Are the rules right? Are there enough of them? Do some contradict others? If expert systems are built which approach the complexity some computer scientists say we can expect in the near future, we should not be surprised at hearing something like the dialog Arthur C. Clark wrote for 2001: A Space Odyssey. In one scene space man Dave Bowman is locked out of the spacecraft by HAL the onboard computer (obviously a PROLOG-based expert system):

Bowman: Open the pod-bay doors, please, HAL. Hello, HAL, do you read me?

HAL: Affirmative, Dave. I read you -- This mission is too important for me to allow you to jeopardize it.

HAL has reasoned that the only way in which he (she?) can complete his mission in space is by killing the crew. It's perfectly clear to HAL even if it isn't clear to the crew. In the end, it is a set of conflicting rules in HAL's programming which drives the computer into an electronic psychosis. We predict that large expert systems will be plagued with the same HAL-like problems well beyond the year 2001.

And what of the Morlocks? They reap none of the benefits of the Eloi when it comes to programming ease. This means that even in 2001 someone will still have to bang the bits. Expert systems may become great at diagnosing blood diseases, but ask them to write a conventional program and they'll call for a urinanalysis. Thus we see programming remaining a job which will have to be done by humans for some time yet to come.

The final thing to remember about expert systems is that there is nothing magic about them. The concept of an expert system is just another programming technique. It makes some problems easier to solve, but the results gotten by expert systems may be obtained by conventional programming techniques. Often those selling expert systems lose track of this fact.

Well, how about ADA?

If the expert system isn't the software transistor is there anything around which might be? Sadly, we don't see it. There is, of course, work being done on programming languages with one current result being ADA. ADA brings smiles to the faces of a good number of people. In fact, just the mention of ADA during a presentation (with an appropriate roll of the eyes to the ceiling) is guaranteed to get a laugh. Seriously though, ADA does contain some important features which will be needed if we are to migrate to the year 2001.

Starting with the worst, ADA's least attractive feature is its size. This stems primarily from the fact that members

of committees which design languages have never developed an effective argument against the statement "put the feature in -- if programmers don't like it, they don't have to use it." ADA, and its associated environments, are large enough that there will be local experts in the language. People will be skilled in ADA task management, but won't be so hot on ADA I/O. This will be something we'll just have to live with.

Better features include the attempt to make a really portable language. Languages like C have been touted as being portable, but, in fact, most of the portability found is C is a result of the cleverness of the programmer. ADA's portability comes more from within. Portability will be extremely important in 2001. This is because systems built for the Eloi will be quite expensive and it will be important to amortize that cost over a large number of installations. To have a package which runs on many machines will help out.

If language efforts like ADA are making important contributions to program portability, then they are also making changes in people portability. People portability? People portability is being able to get your programmers to easily migrate from one computer to another. UNIX and C have gone a long way to make portable people a reality. If software and programming environments move easily from computer to computer, then computer systems will tend to look more or less the same. "If it's UNIX I can make it work" is something we have heard UNIX programmers say. We will be hearing more of this in 2001.

Portability will be a good thing for programmers and computer customers of the future, but perhaps not such a good thing for computer vendors. If everyone has the same operating system (UNIX?) then computer customers will no longer be held to a given vendor. Customers will be able to "shop" for solutions and buy the most bits for the buck. Vendors will no longer be able to count on the captive customer for their computer sales. They will have to compete through raw horse power or intangibles like support or service.

We will be getting a glimpse of this when HP finally starts selling the Spectrum computer line. The technical computer version of the Spectrum machine will be a UNIX box. This will mean that it will compete with all the other UNIX boxes out there. It will either have to be a barn burner or potential customers will have to believe in HP service, sup-

port, etc., etc. This is dangerous for computer manufacturers, especially for those who don't make their own chips.

ADA and other language systems, as opposed to compilers, will also aid in some of the organizational facing programmers. ADA compilers maintain application data bases which allow routines to be compiled within the context of a given intended usage. This enables the compiler to make more checks to insure that subroutines are called correctly and that parameters are passed as required.

If all of this sounds like Big Brother, you're right. In the future, we predict that much of the romance of programming will be gone. Many of today's software gurus pride themselves in being nonconformists; working odd hours and subsisting on peanut butter cheese cracker sandwiches. "No neckties for me, no sir!" ADA (or at least the intention of ADA) is the beginning of the end for the happy hirsuit hacker. Building for the Eloi will require legions of Morlocks and legions require order not anarchy. Programming as a means of self expression will begin to fade as the programming languages and tools start to insure that you have to play it by the rules. Hackers may hate this, but like the Great American Cowboy, they will have to make way for Big Business. Managers need more control over projects and completed software will have to be easily maintained. Remember, portability will mean that software will have an extended life cycle.

Finally, we are beginning to see techniques and tools emerge which address the design and support phases of software development. The cobbler's children often run bare foot. This is true for programmers. It seems as if they are often the last to benefit from computerization. Work must be done in software prototyping in order to avoid lengthy prose descriptions of what systems will be like. Wouldn't it be much nicer for developers and customers alike if they could sit down at a computer and watch a prototype of the application execute. One terminal session is worth a thousand pages of typed description.

Software change control systems are in use now. We see improvements in them and the integration of program development subsystems. Perhaps expert systems will help us stay on track when managing our time and our work load as programmers, designers and debuggers. In the past we have devoted much of our time to the development of the ideal programming language. Now we are starting to realize that

there may be equally important uses for the computer in other phases of the program life cycle.

And in conclusion....

Users of computers will have a field day by the year 2001. They will be freed from the nuts and bolts of programming, even if they are restricted in what they can use the computer for. Well defined applications will be easily performed by expert systems in areas which will likely surprise us.

There will, however, probably be even more need for classic programming in the future. For those who choose to do this work, we just don't see the software transistor waiting around the corner. The problems of program development are profound and are inexorably intertwined with being human. Tools are under development which will make life a little easier for those who will do "real" programming. Software portability, programmer portability and Big Brother programming environments are all steps in the right direction. We think, though, that that's the best we can hope for: a slow and steady sequence of steps toward Every Programmer's dream -- to put himself out of work.