Elegant Technologies

Peter B. Reintjes

April 3, 1992

Abstract

Since being a circuit designer in the 1970's, I have worked with three technologies that I consider elegant. The first two, CMOS and UNIX, have experienced phenomenal success after considerable initial resistance. The last is Prolog. I would like to develop this notion of Elegant Technologies in an effort to understand the broad range of current and future uses for Prolog.

1 Introduction

There are many technologies for hardware and software development. They have different strengths and weaknesses and we are frequently required to choose between them in order to accomplish a task. I have either been judicious or extremely lucky in the choices I have made. I began by using CMOS technology when few people had heard of it. It was considered to be a specialized technology which would not be widely used. With the phenomenal growth of CMOS IC technology in the 1980's I was in the position of having many years experience with what many considered an "emerging" technology.

Similarly, I began using UNIX in the late seventies, and found such skills much in demand a few years later. In each case, I chose to work with the technology because it greatly simplified my work. While it was quickly accepted in academia, UNIX met with considerable resistance in the computer industry. It ultimately went on to dominate the scientific computing industry.

More recently I have been using Prolog to develop application programs, primarily in the area of Electronic Computer Aided Design (ECAD). The feeling of déjà vu is overwhelming. The power and sense of clarity I get from the technology is matched only by the resistance coming from the mainstream of the programming culture.

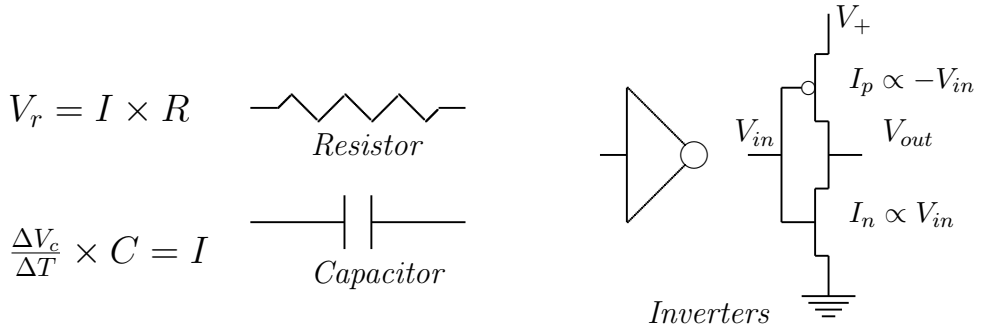
I believe these three technologies share characteristics which many people do not recognize as being important, but ultimately lead to success. These qualities are not disjoint, but seem to come from a wellspring of simplicity and coherence. I have chosen to refer to this as the elegance of a technology. My use of the word is inspired by Richard O'Keefe's dictum that "Elegance is not optional" when writing programs [OKeefe].

2 CMOS

In the old days, circuit design required a wealth of detailed information to be remembered and calculated. Each vacuum tube or transistor in an amplifier had four to six auxiliary components (resistors, capacitors, and inductors) whose values had to be calculated. Furthermore, the relationships between the macroscopic properties of the circuit (gain, frequency response) and the individual component values is very complex.

Some digital technologies, such as TTL and nMOS, require close attention the details of fanout and noise margins. CMOS is largely immune to noise, and each gate output can drive more inputs than nMOS or TTL. Thus, CMOS design can thus proceed with a fraction of the detail required of the other technologies. Furthermore, it's lower power requirement ultimately allowed us to put many more transistors on each chip. The following examples of CMOS circuits can be understood with only a few basic relations.

Figure 1. *Electrical Relationships*



The rightmost diagram shows a CMOS inverter. When the input voltage is high, current flows through the n-transistor to ground and no current is allowed to flow through the p-transistor. Thus the output is low. When the input is low, the p-transistor is on, connecting the output to $V+$ while the n-transistor is turned off. With high and low voltage levels corresponding to the logic levels 1

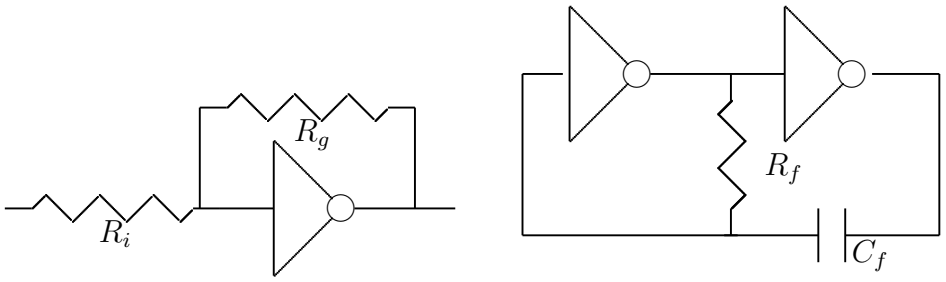
and 0, the circuit clearly functions as a binary inverter.

However, a digital inverter does more than simply invert a signal. For digital systems to work, these gates must move large numbers of electrons in response to relatively small numbers. That is, they must amplify the signal. A CMOS inverter is a highly non-linear amplifier. If the input moves a little in one direction, the output will quickly move all the way in the opposite direction. But despite appearances, we are not restricted to working with binary levels.

2.1 Feedback

Unlike other digital technologies, CMOS can also be used for analog circuit design. For example, if you feed the output of a CMOS inverter through a resistor to its input, the inverter becomes a linear amplifier. This is the simplest imaginable application of negative feedback stablization. As the output tries to go high, the current through the resistor will begin to raise the input, which in turn tends to lower the output. Such a circuit immediately reaches a stable equilibrium at the precise bias point of a push-pull linear amplifier [RCA73]. In other words, the circuit achieves its fixed-point. An oscillator is similar, in that it tries to seek the equilibrium point, but repeatedly overshoots the mark.

Figure 2. *Analog CMOS Circuits*

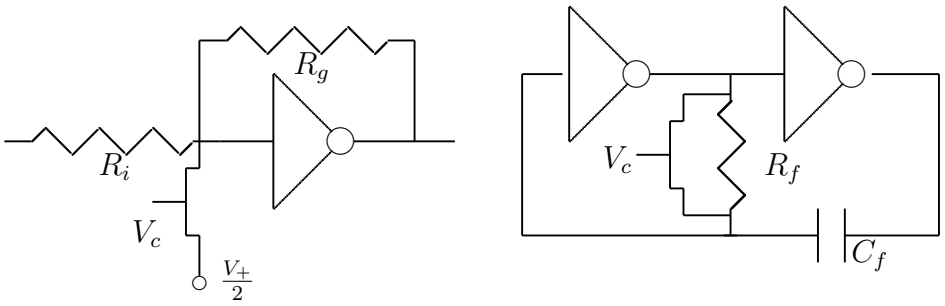


Amplifier $Gain = \frac{R_g}{R_i}$ *Oscillator* $f = \frac{1}{1.4R_fC_f}$

If elegant technologies are characterized by the wealth of function that comes from a few simple structures, CMOS certainly qualifies. Instead of building a linear amplifier by meticulous calculation and component selection, we let negative feedback do all the work. The amplifier is linear because it has to be. Any negative current pulling the input low must be matched by a positive current from the feedback loop. The voltage amplification factor is simply the ratio of the feedback resistor divided by the input resistor.

In figure 2, the component values determine the amplifier gain and oscillator frequency. With CMOS, it is quite easy to modify these circuits to have parameters which vary in response to a control-voltage. In figure 3, the circuits are transformed by the addition of a single n-channel transistor.

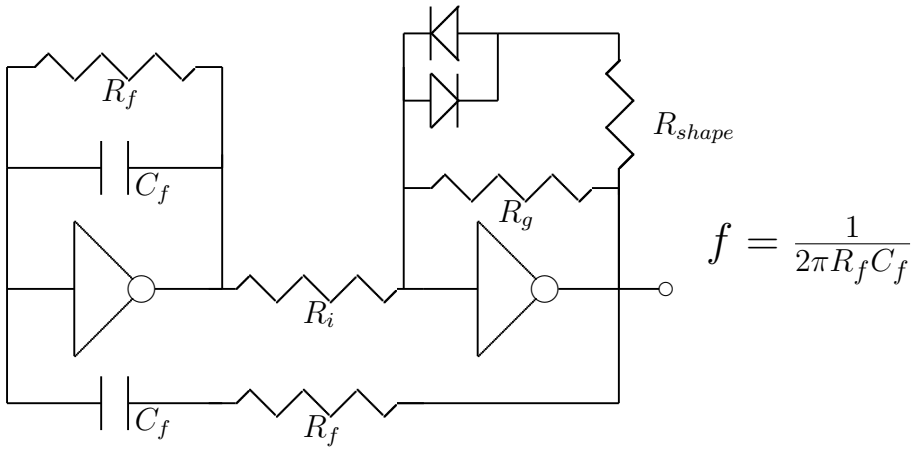
Figure 3. *Voltage-Controlled Circuits*



Voltage-Controlled Amplifier *Voltage-Controlled Oscillator*

Instead of the oscillator's frequency, we can control the pulse-width by connecting the n-transistor to ground instead of the point between the two inverters. For an oscillator that produces sine waves, the Wien-Bridge oscillator below takes advantage of the exponential response of back-to-back diodes. The final shape of the "sine wave" will be determined by the value of resistor R_{shape}.

Figure 4. *Wien-Bridge Sine Wave Oscillator*



As before, this design can evolve gracefully. Substituting different configurations for the diode-pair above will produce other complex wave shapes. Similarly simple circuits can be constructed for analog and switch-capacitor filters and

phase-locked loops such as those found in modern FM radios. The abilities to evolve gracefully and to span a wide variety of applications are fundamental characteristics of an elegant technology. The lack of extraneous components results in circuits which are readable in a way that other technologies are not.

Generally, when two circuits such as an amplifier and a filter have been designed separately, one cannot simply plug the output of one into the input of the other. One must consider loading and impedance matching, possibly needing to redesign one or both of the sub-circuits for them to work together. But these CMOS circuits can be easily re-combined because of the high input impedance of MOS transistors. This ability to combine designs in different ways is also an indication of elegance.

In the early seventies, while working with CMOS IC's, my associates were about evenly divided between the "practical" ones who used TTL chips, and the "forward thinking" ones who were working with nMOS technology. I was pretty much alone in my enthusiasm for CMOS, I couldn't find suppliers who would stock the parts I needed, and was literally ridiculed for my fanaticism in favor of CMOS. The primary objection to CMOS was that it was slower than the other logic families.

CMOS ultimately became the dominant logic technology, and now accounts for the vast majority of integrated circuits. The same symmetry that makes CMOS easy to work with allowed the device scaling which improved its performance. More important than speed was the fact that CMOS circuits used significantly less power and could therefore be packed more tightly together.

3 UNIX

At the same time that I was designing CMOS circuits, I was exposed to computing in the form of PL/1 programming under TSO. On the basis of that experience, I avoided computer programming for the next few years, preferring instead to work with a blackboard.

My impression of programming was that most of my time was taken simply managing the details of the underlying computer system. The simple act of getting data from a file required knowing several levels of system naming conventions. To make matters worse, these naming conventions seemed to depend on meaningless sequences of numbers. A myriad of data formats meant than using two pre-existing programs required the creation and maintenance of intermediate files as well as additional programs for data conversion. Providing an environment in which a single program could run required something like:

Figure 5. *OS/360 Job Control Language*

```
//HASPRJE3 JOB(PN80394),MSGCLASS=A,USER=PBJ043,
// PASSWORD=XYZZY,NOTIFY=PBJ043,MSGLEVEL=(1,1),TIME=33
//DD INPUT1 DSN='XYM345.DEF.A',VOL=833972,SER=034882,DISP=SHR
//DD OUTPUT1 DSN='FU437.TMP1.M',VOL=196644,SER=10247,DISP=NEW
```

After seeing 6th edition UNIX in 1978, I immediately began a career as a programmer. UNIX was not simply an improvement in operating system technology – it completely reversed my earlier decision to avoid a career in computer programming. With a few simple concepts, UNIX obviated many of the time consuming activities of programming. Among these ideas were a uniform hierarchical file system in which I/O devices appeared as files, a convention of treating these files as a simple linear array of bytes, and a standard way to pass data between running processes (pipes). UNIX was more than an operating system and a collection of utility programs. It represented a style of software development that became known as the "Software Tools" philosophy [Kernighan].

I found this working environment particularly well suited to graphics and signal processing. When I had managed to get a digitized version of some songs of the humpbacked whale, I tried analyzing this with an Fast Fourier Transform (`fft`) program. However, due to high-frequency noise, I found it necessary to write a low-pass filter in C.

Figure 6. `lowpass.c`

```
main()
{ float a[3];
  int i = 0;
  scanf("%f %f",&a[0],&a[1]);           /* Read 2 numbers    */
  printf("%f\n",a[0]);                  /* Write the first   */

  while(scanf("%f",&a[(i+2)%3]) == 1) { /* Read a number     */
   printf("%f\n",(a[0]+a[1]+a[2])/3.0); /* Print the Average */
   i = (i + 1) % 3;                     /* Increment index   */
  }
  printf("%f\n", a[(i+1)%3]);           /* Print last one    */
}
```

Now I could send the filtered data to the FFT program with:

```
% cat whale | lowpass | fft
```

I had previously written a program (`muse`) that displayed musical notation on a graphics terminal. It now occurred to me that with the addition of a simple program to convert the strongest amplitude component of the FFT to the nearest note in the equal-tempered scale (`freq_to_note`), I could display the whale's song in standard musical notation:

```
% cat whale | lowpass | fft | freq_to_note | muse
```

In contrast to the complex JCL to run a single program, this command executes and allows data to pass between five programs. But it was not only the behavior of UNIX that was simple and elegant. I will never forget the experience of reading the source code for the sixth edition of UNIX. With the aid of an excellent commentary [Lyons], the ten thousand lines of code could be worked through in detail in about three weeks. Once again, I was confronted by a system which exhibited both remarkable flexibility and comprehensibility.

In 1980 I was working for Data General Corporation and well remember the arguments against the use of C and UNIX. In particular, the primary objection to UNIX was that it was too slow and wasteful of system resources. After all, who had heard of writing an entire operating system in a high-level language. Today, such objections are comical.

What the early critics of UNIX missed was the significance of easing the programmer's task, particularly in regard to portability across systems. They considered only the efficiency of the operating system to the exclusion of programming effort. In UNIX, as in CMOS, we notice the combination of simplicity and power that comes from a few simple ideas, and the resulting benefits of modularity and readability for managing complex systems.

4 Prolog

In the more recent past, a programmer was concerned with memory allocation and deallocation, careful updating of pointers and (if you were really good) the passing and testing of status values to indicate the successful operation of each subroutine. The first thing I noticed about Prolog was that it essentially eliminated these three programming activities, allowing me to concentrate on the problem at hand. I began using Prolog to build tools for parsing, translation, circuit extraction, timing analysis, fault analysis, and interactive editing. The conciseness and clarity that was possible in all of these different areas astounds me to this day.

The ability to define a complete interactive graphical editor in a few thousand lines of Prolog convinced me that a complete VLSI CAD system could be defined

in about 50,000 lines, as opposed to a million lines of C code. Since the sheer bulk and consequent unreadability of traditional software is, or will soon become a primary impediment to technological progress, this compactness is of considerable importance.

Conciseness alone is not a virtue, as anyone who has produced write-only code in languages like APL should know. But well-written Prolog is also remarkably close to natural language, a characteristic which must be considered essential for improved software engineering.

One of the most extraordinary things about Prolog is the wealth of techniques and mechanisms which come out of a few central ideas. In this respect it seems quite different from other high-level paradigms where increased power is accompanied by a profusion of new terminology and support functions. Logic Programming covers an incredible amount of terrain with only a few ideas like unification, resolution, and a simple-minded search procedure. As in CMOS and UNIX, the fundamental units of Prolog programs are small and easy to understand. Also, one finds that larger functional units can be used in a myriad of surprising ways. Prolog is clearly an elegant technology and provides opportunities to make major contributions in Software Engineering, Rapid Prototyping, Formal Language Processing, Simulation, and Formal Verification.

4.1 Software Engineering

The declarative and procedural readings of a well-constructed Prolog program can be very close to a natural language specification. When this combines with the great reduction in Prolog source code size as compared to C, we have an opportunity to radically change the presentation of programs.

All programming languages have an interpretation in natural language, and the naturalness of this translation can be debated. However, if we examine the rules for good prose style, they can be seen as rules for good Prolog programming style. This is true to an extent that is not true in other languages. These rules are taken from *The Elements of Style* with "code", "clause" and "predicate" substituted for "words", "sentence" and "paragraph".

Figure 7. *Elements of Prolog Style*

1. Revise and Rewrite
2. Omit needless code
3. Keep related code together
4. Put statements in positive form
5. Choose a suitable design and hold to it

6. Make a predicate the unit of composition
7. Express co-ordinate ideas in similar form
8. Work with nouns and verbs
9. Use orthodox spelling
10. Be clear
11. Prefer the standard to the offbeat
12. Do not take shortcuts at the cost of clarity

One reason for this closeness between the values of good writing and good Prolog programming is the correspondence between clarity and efficiency that Prolog exhibits. It is more difficult to forsee improved software engineering in other languages where clarity and efficiency are often at odds.

Traditional program documentation hides or ignores the volume of detailed code in an effort to communicate the main ideas. Programs with such superficial documentation are not maintainable. The much smaller Prolog program can serve as the backbone of a document which describe the issues and complete implementation details of a software product. Every line of code should appear in this document. I disagree slightly with Knuth's technique of "transforming" a document into program source (with possible reordering) and suggest instead that the organization of the document should be identical to the organization of the program. This presents a challenge which should be met rather than avoided. The purpose of documentation is to understand a program. While the final organization might be motivated by either programmatic or documentation considerations, it is important that the result is a single entity for which compilation and typesetting are two simple transformations. Moving between the documentation and the program should require as little effort as possible.

But what of other languages which allow for the elegant expression of algorithms? For example, APL clearly has algebraic elegance. The difference is that the typical program a few decades ago was 80% algorithm and 20% user interface, whereas today these percentages are reversed. Prolog exhibits elegance in parsing and database access, and these aspects of programming, and not numerical algorithms, related to the user interface. Prolog goes beyond algorithmic elegance by giving us a tool to handle even command-line arguments elegantly and to allow elegant documentation.

The compactness of Prolog makes possible a style of *total* documentation, where every line of code in a program appears in the documentation. As in good writing, where every word is important, every goal in a good Prolog program has an essential purpose that is vital to the program. A document which hides or glosses over details is not useful for maintenance, which is the dominant cost of software development.

If the discussion of a particular predicate really intrudes in the discussion of a program, then it isn't a fundamental part of the program. Perhaps it belongs in a library (where it will be documented) or, better yet, turn out to be unnecessary. One frequently learns something profound when failing to fit an awkward fact into an otherwise coherent theory. Writing and re-writing programs will become the norm rather than the exception.

4.2 Rapid Prototyping

If, because of the subtlety of Prolog, it takes longer to write an efficient program than in C – how can Prolog be good for rapid prototyping?

Rapid prototyping is concerned with producing an executable specification, a process which is less concerned with efficiency and robustness than production program development. However inefficient or poorly constructed, an *executable* specification of a software product is qualitatively different from a paper specification. The task of turning a working program into a good one is considerably easier in Prolog than in C. This ease is a natural consequence of the ten-fold reduction in source code size and declarative semantics of Prolog. In contrast, think how infrequently anyone bothers to turn a working program into a good program.

Once an algorithm and data structures have been selected, the process of producing a good Prolog program from a poor one is almost formulaic [OKeefe]. One makes the program as deterministic as possible while simultaneously eliminating cuts and removing uses of `assert`, `retract` and `append` by adding state and accumulator arguments.

Constraint Logic Programming (CLP) languages represent an even greater improvement to rapid program development. With CLP, programmers can use the abstraction of constraints to take advantage of linear programming [Lassez] or graph-theoretic algorithms [Carlsson]. Not only is it unnecessary for the programmers to understand these algorithms, they might be completely unaware that the constraint system is using them.

4.3 Formal Language Translation

Natural Language translation is hard. This is primarily because of problems with ambiguity and the requirement for a large amount of world knowledge. While formal language translation is far from trivial, it does not share these fundamental difficulties. The primary functions of a translation system are parsing and database access, two technologies which are a fundamental part of Logic Programming systems. Not inconsequentially, this kind of translation is a major

problem at many levels of the multi-billion dollar ECAD industry.

The first Prolog-based tool to directly address multiple-language translation was AUNT (*A Universal Netlist Translator*)[Reintjes]. This program contains five parsers and generators for hardware description languages and an internal logical form common to all of these languages. Thus, the system implements 20 virtual translators. This work has been extended in MULTI/PLEX, a program which automates the creation of complimentary parsers and generators from a single grammar specification.

4.4 Simulation

In this section, I examine several ways in which Prolog can be exploited to improve Circuit Simulation and Formal Verification. In particular, the conclusion that Prolog is unsuitable for numerical processing is challenged by pointing out how Prolog can be viewed as a compiler technology for numerical processing [Clocksin88].

Today's fastest electronic circuit simulation programs are primarily compilers which produce executable code corresponding to the elements of a circuit. The resulting program is a software version of the design where procedure arguments correspond to electrical connections. A simulation consists of executing this program with input and output data representing the time sequence of data values for the "wires" connecting the circuit to the outside world. A simulation engine is a set of skeletal subroutines which are called by these code fragments as they execute. Subroutine preambles and postambles manipulate data stacks similar to the activation records of computer programming languages.

Given that modern simulation is primarily the execution of compiled logic circuitry, more efficient simulation will depend upon increasingly sophisticated compilation. An effective simulation environment must support incremental dynamic loading of this code and complex bookkeeping to incorporate and resolve links to modified sub-circuits.

These and many other features of modern simulators already exist as fundamental features of Prolog systems. At a more basic level, the similarity between compiled logic circuitry and compiled logic programs has not gone unnoticed [Clocksin87]. The ability of Prolog to backtrack adds even more power to examine and re-execute simulation steps. In recent years, there has been interest in non-deterministic hardware models, for which Prolog is a natural medium of expression and simulation.

Therefore we see Prolog technology as being an appropriate tool for building more efficient and more flexible simulators. Not necessarily by writing simulation algorithms directly in Prolog, but as a tool for more efficient compilation of circuit

descriptions into machine language.

4.5 Verification

As system complexity has increased, we have passed the point where entire systems can be verified by exhaustive simulation. The most promising alternative is formal verification, which consists primarily of proving that an implementation is functionally equivalent to a specification. Without formal verification, the situation is analogous to a mathematics without proofs, where all variables must be reduced to arithmetical quantities and the arithmetic carried out. Several hardware proving systems exist but can only be used by people skilled in theorem proving.

Computer-aided design provides an interesting opportunity in formal verification. Unlike theorem proving or post-hoc software verification, we can create design languages and design processes oriented toward formal verification. Incremental verification, dovetailed with design steps, can eliminate the combinatorial explosion associated with most forms of verification. The challenge is to provide tools for incremental verification that do not require designers to have a grounding in proof theory. Failing that, we might attempt the creation of a framework in which designers can more easily learn the techniques of formal verification.

While Higher Order Logics (HOL) [Gordon], are currently the most promising approach to post-hoc formal verification, the "correct construction" approach can probably be implemented in less powerful proof systems.

4.6 Summary

Software Engineering – Prolog implementations of programs yield an order-of-magnitude less source code and a qualitative difference in readability, and hence software maintainability.

Rapid Prototyping – Prolog offers a framework for creating executable specifications of programs. Constraint logic programming extends this by providing numerical and graph-theoretic algorithms under the simple abstraction of constraints.

Language Processing – Prolog is well-suited to formal language translations for the same reasons it is used in the considerably more ambitious task of natural language processing.

Simulation – Prolog's suitability for compilation [Warren] and optimization

[Clocksin88] make it a good framework for building native code compilers for simulation languages.

Verification – Prolog's weak form of theorem proving should prove sufficient for incremental verification of systems which are constructed with provability in mind.

5 Obstacles

Balancing this optimism are several obstacles to the exploitation of Prolog technology. Most notably, these are the lack of software maintenance metrics, various performance issues, cost, and academic as well as industrial education.

5.1 Lack of Software Metrics

The development of a software product can often be measured in the sense that a manager knows how many people were working for how long between the conception of a product and its first release. At this point accountability becomes difficult. Software maintenance of a released product often requires cycle-stealing from developers working on the next product, and bug fixes are interleaved (or confused) with feature enhancement. Subsequent releases can involve large numbers of employees who are not acknowledged as being directly responsible for a software product.

Maintenance is estimated to be well over half of the total cost of software. In advocating the use of elegant and powerful tools such as Prolog, we may initially increase the more easily measurable costs (tools, training, delay in development) in an effort to decrease a larger, but less well monitored maintenance cost. Unfortunately, short-sighted managers will prefer to reduce the measurable expenses for which they can be held directly accountable. Thus, the absence of maintenance cost measurement is a major impediment to improved software engineering.

5.2 Performance

People frequently make the comment that Prolog is slow. There are many ways to address this criticism, but first, one must determine whether the performance problems are the result of inefficient programs written by novices. In most languages, poor programming style may reduce program efficiency by a factor of two. In Prolog, which allows abdication of control (but doesn't require it!), naive programs can be orders-of-magnitude slower. Thus, the extra power of Prolog, in the hands of a novice, has an extra cost.

While performance is often given as the primary objection to Prolog, there are many reasons to be optimistic.

- Many performance problems are the result of poor programming.
- Nothing is cheaper than computation. Price/performance has improved a billion-fold in the last forty years.
- Prolog compilers continue to improve. Recent work with abstract interpretation/global analysis indicates a minimal twofold performance improvement [VanRoy].
- Much numerical computing can be subsumed by constraints, allowing Prolog programmers to transparently employ numerical algorithms implemented in low-level languages.
- Prolog is a powerful technology in which to construct specialized compilers to generate imperative code from high-level specifications.

5.3 Cost

Prolog has an image problem associated with the cost of industrial implementations. By describing Prolog development systems as "compilers", vendors encourage customers to compare these systems with C or C++ compilers. Managers who have no idea what Prolog is, would have to conclude that it is too expensive.

Vendors need to emphasize that Prolog systems contain a relational and deductive database system, a memory management system, an inference engine, a backward chaining expert system shell, a parser generator and a theorem proving, rapid-prototyping, interactive development system with incremental loading. Perhaps they should also mention that a compiler is included, at no extra cost.

5.4 Education

Academic

In some quarters, academia seems to be retreating from the cutting edge of programming language awareness. While excellent work continues at the graduate-level, very few undergraduates are being exposed to Prolog or LISP. The current trends in programming have convinced academics that C must be taught, to the exclusion of other languages.

Such people seem to have no comprehension of how easy it is to learn a language like C or how detrimental it may be to programming style. They underestimate the value of exposure to radically different languages, perhaps thinking that in teaching Pascal, Fortran, and C, they are exposing their students to three important languages, rather than a single language model.

Industrial

Industrial education usually meets with limited success for a number of reasons. Once out of school, people have less interest and energy to devote to new ideas and companies fail to provide motivation comparable to grades in academia. Perhaps more importantly, managers are often unable to give employees the necessary time and resources to develop new skills. This is particularly true with the current trends to reduce work forces.

Luckily, elegant technologies attract intelligent people. Such people will always manage to find the time to develop the necessary skills. But there will always be too few of them.

6 Why do Elegant Technologies Succeed?

One may not immediately associate elegance with usefulness. But for a *technology* to be elegant, it seems reasonable that we require it to have significant practical value. Unless a technology fills a need, all of its elegance is academic. The earlier needs for levels of abstraction in hardware design and systems programming (met by CMOS and UNIX), were similar to the current need for a unifying technology for applications programming. In all three cases, the technology provides a way of dealing with the difficulties associated with size and complexity. In the case of CMOS, it was the ability to design and then fabricate monolithic systems of over one million transistors. For UNIX, it was the ability to transport an operating system and a large body of software, not merely to a number of different machines, but to virtually all machines. Logic Programming languages provide a level of abstraction which greatly reduces the source-code size of programs, thus improving our ability to read, maintain, reuse, and verify them.

Proper etiquette usually requires us to make comments about the complementary nature of different paradigms and to acknowledge that different paradigms provide "the right tool for the job" in different instances. However, there are indications that Logic Programming is superior for symbolic applications, simulation, database technology, translation, financial modeling, knowledge engineering, CAD, theorem proving, and even as a framework for numerical processing.

Have I left anything out?

We are selling Logic Programming short if we only think of it as being appropriate for Expert Systems or other AI domains. My experiences, and the rarity of truly elegant technologies, tell me that Logic Programming could easily be the dominant programming paradigm in scientific and industrial applications by the end of the present decade. There are many reasons to be optimistic about achieving the opportunities I have mentioned. In view of the history of CMOS and UNIX, I even take some comfort from the fact that Prolog is considered to be too slow.

7 Conclusions

- An Elegant Technology is one which introduces a significant level of abstraction with a few concepts.
- Elegant Technologies transcend specialization and exhibit a tendency to do everything well.
- Designs in Elegant Technologies can evolve gracefully.
- Designs in Elegant Technologies are readable, facilitating education and maintenance.
- Early recognition and appreciation of Elegant Technologies requires a value system that places a premium on simplicity, elegance, and comprehensibility.
- CMOS, UNIX, and Prolog are Elegant Technologies.
- Elegant Technologies are ultimately successful because they improve our ability to manage complexity.

Bibliography

[Clocksin87], W.F. Clocksin, "Logic Programming and Digital Circuit Analysis", *The Journal of Logic Programming*, North Holland March 1987

[Clocksin88], W.F. Clocksin, "A Technique for Translating Clausal Specifications of Numerical Methods into Efficient Programs", *The Journal of Logic Programming*, pp 231. North Holland September 1988

[Carlsson], Mats Carlsson, "Boolean Constraints in SICStus Prolog", *SICS Technical Report*, T91-09, September 1991

[Gordon], Michael Gordon, "Why Higher Order Logic is a Good Formalism for Specifying and Verifying Hardware", *Formal Aspects of VLSI Design*, edited by G. Milne and P.A. Subrahmanyam, North Holland, 1986

[Jaffar], J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proceedings 14th POPL* pp, 111-119, ACM, January 1987

[Kernighan], B. Kernighan and P.J. Plauger, *Software Tools*, Addison Wesley, 1978

[Lyons], John Lyons, *A Commentary on the UNIX Timesharing System*, University of New South Wales, suppressed document

[OKeefe] *The Craft of Prolog* MIT Press 1990

[RCA73], *COS/MOS Digital Integrated Circuits* RCA Solid State Databook Series, SSD-203A RCA Corporation, 1973

[Reintjes], "AUNT: A Universal Netlist Translator", *Journal of Logic Programming*,8 pp5-19, North Holland 1990

[VanRoy], Peter Van Roy, "High Performance Logic Programming with the Aquarius Prolog Compiler", *IEEE Computer*, pp 54-67, IEEE January 1992