TECHNICAL FLUENCY STIFLED FOR DECADES

Edward S. Lowry
Bedford Mass.
eslowry@alum.mit.edu
http://users.rcn.com/eslowry

Draft 8 January 2010

For 35 years, prestigious organizations have collectively obstructed progress in technology for simplifying the expression of software and other technical knowledge. They have avoided noticing:

- **their estrangement from traditional engineering values such as identifying and advancing the leading edges of simplification.**
- **a poverty of available expertise and literature on simplification.**
- **their collective silence on some central issues of information technology.**
- **that pieces of information can be designed so they can be easily arranged.**
- **that students are massively taught how to arrange pieces of information by teachers who have little idea what is a reasonable design for pieces of information.**
- **large opportunities to simplify technical information.**
- **that they have been profitably forcing millions of software developers to generate complexity that broadly undermines software quality.**
- **that eliminating harmful complexity from computer software is largely the same problem as its elimination from other technical knowledge.**
- **the potential for unification of semantics for precise language.**

Knowledge of simplification can show providers of technical information how to be more helpful and less controlling toward others. First steps in resuming progress could focus on the candor of those who participate in the management of safety-sensitive complex systems.

Those who profit by coping with complexity often have incentives to keep things complicated and obscure the potential for simplification. Deliberate, high level decisions have effectively restricted software to be developed using language technology with poor simplicity of expression compared with what was operational 25 years ago. The result has broadly impaired the reliability of software that controls safety critical systems. One reason for concern is highlighted by the overheating and emergency shutdown of a nuclear power reactor near Baxley Georgia in March 2008 caused by a software update.

For roughly the last 40 years, the number one dominant technical problem in computer science has been, or arguably should have been, reducing needless complexity. A direct approach to progress has been to identify various sources of needless complexity and eliminate them. The one dominant obstacle to such progress has been prestigious organizations and leaders who do not want their needless complexity eliminated. The problem is illustrated by masses of needless complexity on display in the computing section of large bookstores. Much of that complexity has the effect of

Part of this article is reproduced from: E. S. Lowry, “Software Simplicity, and Hence Safety – Thwarted for Decades” in ISTAS04 © IEEE 2004

turning people into generators of more needless complexity. The same kind of needless complexity degrades technical communication with both people and computers. The efficiency of mental effort in working with technical knowledge has been widely impaired. There is no sound assessment of how crippling this deprivation of technical fluency may be.

As sources of needless complexity are identified, the remaining problems become clearer, making it easier (technically if not politically) to further reduce complexity burdens. The burdens include five basic kinds of "Inexcusable Complexity"[1]. They are described briefly in Appendix A below. Fixing the first problems makes it easier to fix the others. There are tradeoffs in evaluating some kinds of complexity such as syntax, but in the underlying language semantics, needless complexity is more purely harmful. When eliminating needless complexity, differing quantifications of complexity have little effect on design decisions so that progress in advancing the leading edge can avoid ambiguity. The literalness of computers sharpens the analysis. When people follow oversimplified instructions, they often muddle through, but not computers.

Today, I doubt that any organization is seriously pursuing simplicity of expression in computer language as advanced as that provided in a design distributed at IBM in 1974. The world's best facilities for executing simply-expressed software are now substantially less capable than what was operational at Digital Equipment Corporation 20 years ago.

For all practical purposes, the software leadership has collectively opposed large scale simplification over the past 35 years. Software quality has been undermined in foreseeable ways by deliberate decisions which have caused a widening swath of death, destruction, ignorance, agony, waste, criminality, and dangers to national security. There is hard evidence [2] supporting the fact that the delay has occurred. Since the 1970s, human-machine interactivity and easy access to large amounts of information have improved, but progress in composing software has largely stagnated. In the past decade, the Science News year-end summaries of science news report only peripheral developments in software technology.

Vested interests in complexity can be tenacious. It took centuries to switch from Roman numerals to Hindu-Arabic numerals. It took a full blown revolution to simplify the French legal and measurement systems. Three mechanisms may contribute to the tenacity. First, when clients trust providers (including employees etc) to deal with complexity, the providers may find they have incentives to resist simplification or add to the complexity because that gives them control over their clients and job security. Second, succumbing to such temptations creates incentives to be less than candid with their clients about the practicality of simplifying. It can happen to individuals or large groups such as the legal profession. Complexity can undermine client relationships that require trust, especially when prestige of information providers is at stake. A third mechanism is that the degraded relationships cause cognitive dissonance which gets resolved when the providers make themselves oblivious to the possibility of significant simplification, enabling them to think of their behavior as honorable. The inattention to large scale simplification by technical information providers seems to fit this model.

While simplification can enhance autonomy, it can also undermine traditional interdependent relationships in the process. Providers depend on having customers who depend on them. Employees depend on having employers who depend on them. Teachers depend on having students whose learning depends on being taught by teachers. The prospect of large scale simplification can, and apparently does produce understandable anxieties about those relationships. Concern for the relationships is legitimate but not responding in destructive ways.

Obstructions

For 2008, IBM reported 85% gross profit on $22B software revenue. Few legal businesses aspire to profit margins that large. To sustain that profit level, there is incentive to keep software complicated. Doing so makes switching vendors impractical so that market competition will not work to reduce profit margins. When I started showing people at IBM how to make substantial simplification to software in 1972, senior people reacted with hostility even though the company had just started a billion dollar project whose stated goal was to make development of computer applications easier.

In 1975 I went around IBM asking what better way there was to achieve the project goal than my approach to simplifying software. One mid-level staff man shouted lustily “I won’t answer that question. You can’t make me answer that question”. Another representing the Board Chairman repeatedly said “I will not entertain that question”. From that dubious level, candor on the subject of software simplicity has broadly gone down hill since. In 1976 IBM refused a request to release the shelved technology to the US Department of Defense. In 1991, 20% of identified US casualties from the Gulf War were associated with a software failure.

A large fraction of the world’s workload is arranging pieces of information. Arranging them would be easier if the pieces are designed to be easily arranged. Designing them that way is an obvious thing to try, but for over 20 years software leaders and others have obstructed such efforts. Few people have seen basic building blocks of information that are well designed to be easily arranged in rich information structures. Currently used pieces of information have traditional structures (text, tables, bits, records, etc) which lack serious optimizing design. A number of computer scientists have spoken as though it were an accepted convention among computer scientists that they do not, or will not further consider questions about the design of pieces of information. In 1988, I was instructed by management not to discuss such design in my work at Digital Equipment Corporation and a senior computer scientist summoned me to his office at Brown University to warn me of the futility of trying to publish on the subject.

The biggest disaster is probably in technical education. Students everywhere are routinely taught how to arrange pieces of information by teachers who have no real understanding of the design of easily arranged pieces of information. The result is a massive failure to express precise information in precise ways, leaving students mired in mind-hobbling vagueness and complexity [3].

Cyber security is another disaster area. Software systems are full of security holes because they are full of bugs because they are full of worthless complexity whose production should have been

banished 30 years ago. The cyber security community is oblivious. In early 2009, the Institute for Information Infrastructure Protection released a report "National Cyber Security Research and Development Challenges". Ninety-two experts made twenty-seven recommendations making no suggestion that anything needed to be simplified.

Hardly anyone overtly opposes improving simplicity of expression in computer language, but a significant number overtly oppose wide generality of computer language. Opposing either is equivalent to opposing both. Providing greater simplicity of expression in a language also provides greater language generality because there is then a wider class of applications for which its simplicity of expression is adequate. In July 2008 both leading computer science magazines Communications of the ACM and IEEE COMPUTER included articles [4,5] which claim that we "must" avoid one-size-fits-all solutions. The cultural disapproval of trying to broaden language generality tends to be out in the open, but without acknowledgment of implications for complexity. The disapproval is largely unchallenged and expressed without regard for what is or is not technically practical. Language with substantially improved generality was operational 25 years ago, although used by only a few dozen people. When language generality is improved, it becomes a leading edge of the technology and helps to sharpen other leading edges by providing more natural standards for comparison. Of the five kinds of "inexcusable complexity" mentioned above, lack of language generality is the only one that gets attention in the technical literature, and almost always to discourage working on it.

A few computer scientists have suggested ML as a language which provides good simplicity of expression. I re-expressed a textbook example of an ML program [6] into a program one third the size. Microsoft denied my claims of its backwardness at a court hearing in 2004, possibly a criminal offense. Digital Equipment Corporation spent 9 years pursuing a precedent-setting patent on well designed pieces of information. The main effect has been to further obstruct the use of well designed pieces of information.

The economy of developed countries depends increasingly on "innovation" which depends on creativity which is stifled by needless complexity. The following expressions are expressed in a general purpose programming, data base, and modeling language that was implemented 25 years ago:

- count every person whose spouse is veteran,
- sum revenue of every year after 1996,
- every element where some isotope of it is stable .

This kind of fluency is part of everyone's heritage from early childhood on, but it is not included in currently available precise languages (with the possible exception of some that are highly specialized). Denying that fluency to people when they are trying to be precise has no technical justification and probably undermines their creativity significantly. High stakes decisions are being made in contrived ignorance.

The office of US Senator John Kerry helped to check on the delay. The main question raised was whether anyone working in the US Federal Government had experience working with technology

which allows for simplicity in expressing software as advanced as what IBM published in 1977[2]. From 2000 to 2004 he presented the question to 9 government agencies. None has reported finding anyone that advanced and all of them avoided answering the question.
The agencies queried were:

- Office of Technology Policy (Dept of Commerce)
- National Science Foundation
- United States Air Force
- General Accounting Office
- National Aeronautics and Space Administration
- National Institute for Standards and Technology
- Dept of Education
- Dept of Homeland Security
- Dept of Defense

Only NSF, GAO, NASA, NIST, DHS, and DoD responded on paper. My interpretation of those responses and non-responses is that they could not readily find such competence, they are not interested in developing such competence, and they seemed willing to conceal its absence. To my knowledge there are only a few dozen people anywhere with such experience. They worked at Digital Equipment Corporation during the 1980s. The correspondence shows that a small but somewhat independently selected sample of software leaders were consistent in resisting the truth about simplicity.

The ACM Computing Surveys V28 No1, March 1996 [7] contains "69 short articles that span the discipline of computer science". It almost totally ignores simplicity issues. I found only three sentences which were on topic, and they only acknowledged simplicity as a valid goal. Thousands of highly intelligent computer scientists have somehow been deterred from seriously examining the most fundamental issues in their field. However discomfiting the above perceptions may be, their accuracy is rarely questioned.

Harm Done
Results of decades of deliberate opposition to simplification include:

- Deficient math and science education due to failure to express precise information precisely.
- Users and students entangled in proprietary complexity for decades.
- Massive degradation of the quality of technical information: accessibility, usability, clarity, interoperability, ductility.
- A probable contributing factor to the crash of KA801, August 1997, 226 dead.
- A possible contributing factor to 20% of US casualties in the Gulf War as known in 1991.
- A probable contributing factor to 3 friendly fire deaths in the Iraq War.
- A contributing factor to the August 2003 northeast power outage, 50 million affected.
- A contributing factor in many cyber-attacks.
- Failure of the FAA to upgrade its air traffic safety systems.
- Large part of the $60B annual cost to the US of software errors (as estimated by NIST).

- Political leaders dependent on technical advisors who are over 35 years behind the leading edge on simplicity issues.
- Burdensome technological instability.
- Large learning loads for skills of ephemeral value.
- Computer science research blunted.
- Monopolization in the software industry.
- Multi $B information system fiascos. E.g. IRS.
- Prestigiously sponsored illusions of efforts to simplify.
- U.S. government priorities thwarted in
    - technical education
    - economic innovation
    - cyber security
    - health care costs
    - government spending.

Leaders have tended to stonewall discussion of simplicity issues. Curiosity about whether the claims of a long delay are true does not arise. Interest in showing that the claims are wrong does not arise. Pride in demonstrating capability in the area does not arise. The possibility that the claims might be true is only occasionally acknowledged but quietly. For prestigious institutional representatives, evading the issue seems to be a priority. Sometimes they make vague claims that proficient people are dealing with the problem and there is no need to say more. DoD, IBM, Microsoft, MIT, and NSF have reacted in this style. The response from DoD obscures the cause of death of some American troops.

Estrangement from Engineering

Software technology has not internalized traditional engineering values very effectively. The basic engineering value of getting more for less requires some quantification of benefits and costs, or at least dependable comparisons.

Simplification fits the engineering paradigm better than most computer science. Making data objects more uniform, and up to a point simpler, produces improvements in:

- simplicity of expression in applications,
- simplicity of language, for a given level of capability,
- generality of language

all at once. It expands the engineering envelope on three leading edges. In doing so it presses the limits for those leading edges and clarifies what is possible for each of them. The long absence of disciplined effort to thoroughly remove needless complexity has obscured both the leading edges of software technology and the relevance of traditional engineering.

An example of inadequate use of engineering criteria occurred in the design of the Ada language. It was designed to meet requirements specified in terms of features rather than capability levels which could be compared. As a result Ada has many features but poor simplicity of expression compared with what was known several years before it was designed. Other languages such as

C++, Java, and C# have been introduced with capabilities far behind known leading edges of the engineering envelope. The term "software engineering" has been widely inverted so that its practitioners are more focused on adapting people to technology rather than vice versa.

A highly influential paper, "No Silver Bullet..." by F.P. Brooks Jr. [8] gave poorly justified discouragement to the use of traditional engineering approaches in the improvement of software technology. It included a claim that “No facilitation of expression can give more than marginal gains”. That erroneous, unsupported, and 23 year old statement may be the closest thing in the technical literature to an assessment of the potential for improving simplicity of expression.

In many engineering fields, prizes for winning competitions and breaking records have helped to focus attention on how to advance the leading edges and to create institutional knowledge of best practices. In contrast, the ACM International Collegiate Programming Contest has impeded progress by specifying the poor technology the participants are required to use.

Impressions

The following are impressions based on experience whose detail would be impractical to recount in this paper. The widespread, long term lack of candor about simplification has been striking. Part of it is documented in the correspondence with Sen. Kerry’s office. People seem to be consciously (but not openly) making choices between being helpful with simplicity and being controlling with complexity. In technology businesses, control of clients has become highly prized. Government officials and professional societies also seem to lack candor about simplification but for additional reasons like protecting employment.

The willingness of academics to impose mind-hobbling complexity on their students has also been striking. It has been particularly noticeable at prestigious universities including some faculty who describe their goals as humanitarian.

For decades, leaders and forums have shown little tolerance for efforts at large scale simplification. The aversion seems widespread in software culture and could persist for decades more if not adequately counteracted. The existence of a scientific community being so opposed to solving its most basic problems seems rare and perhaps unique. The few software leaders who publicly comment on simplicity issues seem to say enough to sustain an illusion of sympathy toward simplicity – and little more. Many people seem aware of the ethical issues. There is some willingness to listen to the story of obstruction of simplification but not to speak about it.

The issues are not technically difficult.

Some additional observations:

- The National Science Foundation used misleading evasion in 1996 to avoid exposing its weaknesses in simplicity and again in 2002.

- In 2001 the World Wide Web Consortium volunteered that the languages they were introducing were basically 1970s technology, a small blip of candor.
- In 1992 Digital Equipment Corporation moved the leading edge of its software technology backwards by 15 years from a simplicity point of view.
- The technical journals of the leading computer professional societies, ACM and IEEE, have published nothing on simplicity of expression in computer language as advanced as what IBM [2] and DEC [9] published in 1977 and 1978.
- The IEEE misleads the public and violates its own code of ethics by granting "Certified Software Development Professional" credentials without disclosing its 30 year skill gap in simplicity.
- "Simplicity is a jihad" for Microsoft, said Bill Gates in 1998 [10], as Microsoft prepared to introduce a major new programming language C#, whose simplicity of expression was decades behind the leading edge.

Resuming progress

The technical direction sketched in Appendix A seems appropriate. There remain many people who have serious technical problems needing simplification and efforts to find them may still be productive. Wider knowledge of the obstruction may make them more visible. Better awareness of the long term disregard for public safety by prestigious software organizations may elicit responses.

Organizations which manage complex safety-sensitive systems seem consistently decades behind the leading edge in expressing software simply. They may have the capability to develop such specific competencies as needed, but they tend to brush aside suggestions that they lack significant competence. Intuitively, any organization entrusted with safety-sensitive responsibilities owes the public some candor about their relevant competencies. First steps in resuming progress could focus on the candor of those participating in the management of complex safety-sensitive systems, including those in advisory or regulatory roles. Questions like those below can be asked.

Needless complexity in software contributes to failures in complex systems even when software is not directly involved in the failure. Software can raise the overall complexity and reduce attention to other development issues.

Whether resistance to simplification arises from vested interest, inertia, embarrassment, or unawareness of the delay, public exposure can provide incentives for improvement. While resistance to major simplification is widespread among technical information providers, its depth seems variable and it may be fragile if more strongly challenged. Getting the issues ventilated in appropriate forums is a continuing problem.

Substantial precautions may be needed to prevent the degradation of simplicity goals in any major language development effort.

If the right questions are asked and answered honestly, little technical expertise will be needed to see that prestigious organizations are far from competent in expressing software simply. A pointed line of questioning which almost anyone can pursue is: “Architects and builders tend to know something about the shape of reasonably designed bricks. What do you know about the design of easily arranged building blocks of information? Can you show that your efforts to simplify are appropriate to helping people rather than controlling them?” There is background in Appendix A.

Publicly visible answers to questions like the following could help assess deficiencies, suggest ways to revive progress, and stimulate constructive action.

1. What organizations are competent to represent precise information with simplicity that is appropriate to societal needs in any of:
   - technical education?
   - safety of nuclear reactors?
   - safety and dependability of air transport?
   - security of personal data?
   - safety and costs of software for health care?
   - worker productivity?
   - ease of computer use?
   - improving creativity and innovation?
   - avoiding economic crises?
   - parallelizing software to exploit multi-core computing?

2. What technical leaders have demonstrated curiosity and candor about simplicity of representing precise information that is appropriate to any of the above needs?

3. What political leaders have advisors who are competent to prioritize investments in meeting the above needs?

4. What organizations have recent working experience with computer language which provides for expressing applications as simply as the PROSE language[2] published by IBM in 1977 and operational at DEC by 1983 (under the name Dawn)? If any, using what language technology? There is a simple preliminary test that can check a language's capabilities in Appendix B.

5. What organizations represent richly structured precise information using building blocks of information which are well engineered to be easily arranged?

6. What long term technical reasons are there for using basic building blocks of information other than hierarchically interconnected pointers for representing the data in rich computer applications? See “Toward Perfect Information Microstructures” [11].

7. What technical obstacles could prevent development of a truly general purpose computer language semantics?

8. What organizations have profited by suppressing technology they have developed more than IBM?

9. What other branches of engineering have focused so much on adapting people to technology rather than technology to people?

10. What organizations that depend on or produce safety-critical software are willing to tell the public how well the simplicity of expression of their software technology compares with what was operational 25 years ago?

11. What university faculties doing computer science research are willing to tell their students how well the simplicity of expression of the software language technology they are learning compares with what was operational 25 years ago?

12. What large technology organizations have shown willingness to confront any of the above issues in a forthright way?

> My experience so far supports answering each of the above questions with: “None”. Corrections would be welcome.

13. What software leaders could provide good reasons to give any different answers?

14. What technical issues in computer science are more important than managing complexity effectively?

15. How many design activities in the past 40 years have addressed more fundamental problems than the design of the building blocks of information?

16. How much have the capabilities of recent graduates in technical areas been damaged by decisions to avoid simplification of technical knowledge?

17. Among human intellectual skills, how many are more important than being able to read and write precise information simply? How well can “computational thinking” be developed without effective computational fluency?

18. How effectively can improved simplicity of expression enhance creativity and innovation?

19. What communication chasm between providers and consumers of technology has been larger?

20. What factors have deterred computer scientists from exploring simplicity issues?

21. To what extent does software technology have an Augean quality that contributes to reduced enrollments and gender imbalance in computer science?

22. How does the function or dysfunction of software technology leadership compare with that of leadership in other technologies?

23. What occurrences of negative technological progress have been comparable?

24. What public interest organizations, journalists, ethicists, technology scholars, etc might put the obstruction of simplification into a sound perspective and assess any abuse of public trust?

Technical Appendix A - Combining Structural and Functional Expressiveness

Widespread knowledge of simplification methods can lead to healthier relationships between providers and consumers of technical information. Preferences to control the consumers rather than help them would be less acceptable.

The five sources of inexcusable complexity [1] mentioned above are:

- poor structural expressiveness,
- poor functional expressiveness,
- poor language generality,
- poorly designed data objects, and
- a need to compensate for the above by using imprecise rather than precise language.

This describes briefly a key language design concept that allows for simplicity of expression. It shows how initial simplifications can clear the way for others and may help clarify the ways technical communities have reacted.

A substantial computer language should provide for accurate description of both the structure and the function of discrete models of various evolving systems. The structure of a system at a given time may be represented by interconnected "atomic data objects" (or pieces of information) provided by the language. One language design factor contributing to simplicity of expression is to include data structures in the language which are rich enough that they can closely match a wide variety of conceptual structures to be modeled. Languages which do that well can be described a "structurally expressive". Deficiencies in structural expressiveness will require the use of extra data objects to compensate and that will cause needless complexity in programs for processing the data.

Another helpful language characteristic is "functional expressiveness", the ability to write a single expression which manipulates multiple large aggregates of data. That requires the language to have a single dominant style of stepping through the members of sets of objects so that most

iteration code can be defaulted to that style. Such languages (e.g. APL, SQL) have only a small number basic data object structures.

Historically there has been a dichotomy between computer languages with structural and functional expressiveness. Early efforts to achieve structural expressiveness relied on adding additional kinds of atomic data object such as arrays, records, strings, and pointers. Doing so required multiple ways of stepping through sets and that defeated functional expressiveness. A design that could effectively cure the dichotomy was developed 35 years ago, though it was not understood in those terms at the time. The problem can be cured by using only a single flexible pointer-like structure for atomic data objects [Fig. 1]. That structure allows for high quality structural expressiveness and a single dominant way of iterating for functional expressiveness.

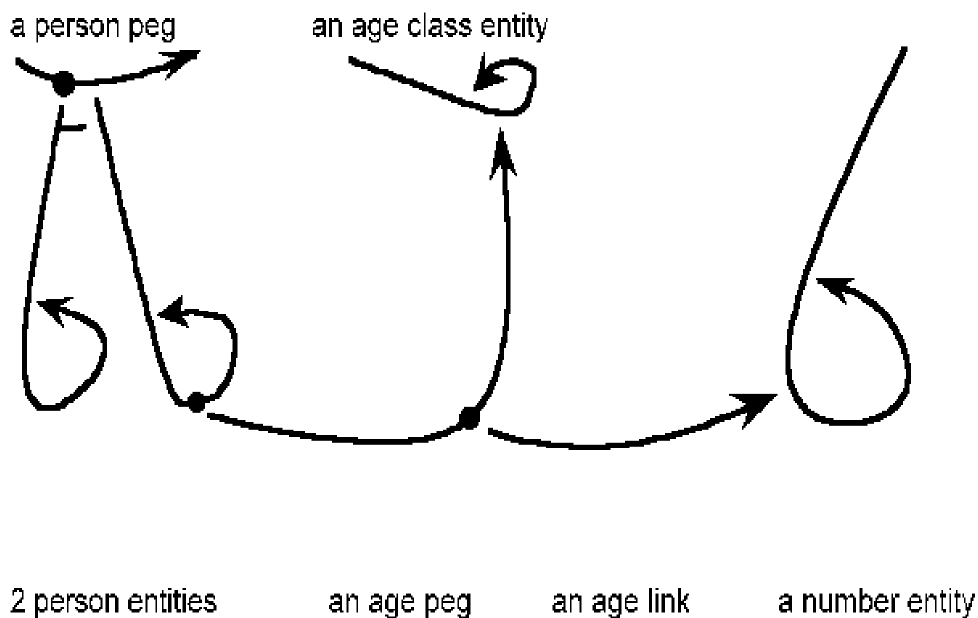

Fig. 1 Well designed pieces of information, representing the age of a person

Fixing the problem allows for wide generality of language semantics in a relatively small language. Specialized semantics could only have advantage where the application has very simple data structures and then only if it is isolated from other applications, and then only if the learning problems are minor. To combine both kinds of expressiveness, the language designer is effectively restricted to choose from only a few practical design choices for the structure of atomic data objects. One of the choices is a single object [11] which is a little better than the other possible choices and it appears able to serve as a practical enduring optimum in non-trivial applications.

No currently used computer languages do as well at eliminating the dichotomy as what was known 35 years ago. Thousands of natural languages have combined structural and functional expressive seamlessly, though not conveniently for precise expression. They all tend to have about

the same levels of structural and functional expressiveness and about the same ability to adapt to any subject matter. The similarity suggests that there a natural limit in those capabilities. When language that does combine structural and functional expressiveness is available it is hard to see a long term future for any substantial language that does not. That can make a lot of current software technology obsolescent and may help explain resistance to simplification.

Providing both structural and functional expressiveness in a single language sets up further simplifications and clarifies the leading edges. It allows for generality of language which can simplify the computing environment. It allows for a practical natural standard basic data object which could reduce interoperability problems. It allows for readable precise system descriptions which can be used in human to human communication with less resort to informal language. Technical descriptions based on current notations use a mixture of precise and imprecise expression which the reader must integrate. Simpler integrated precise descriptions can provide additional options for presentation which are less mystifying.

Appendix B: SHANNON Examples compared with C++

To illustrate the potential for improving simplicity of expression, the following list of example expressions are provided. They can be used as a preliminary test to compare present language capabilities with what was known in the early 1970s. They indicate what degree of simplicity and clarity is achievable in a multi-purpose language and roughly what was known to be achievable in 1974 as recorded in: "PROSE Specification" [2]. It was dated Nov 1977 but it was distributed within IBM in December 1974. A preliminary version was distributed in 1973. The language, later called Keep and Shannon, was implemented and used at Digital Equipment Corporation during the 1980s and then archived. Design work is continuing.

These examples are translated from the first 7 examples given for Sequel2 (now SQL) in the IBM Journal of R&D, Nov 1976[12]. These examples showcase the benefits of including "functional expressiveness" in which iteration code is defaulted and set operations for sorting, averaging, and eliminating duplicates are easily provided. For the first 7 expressions C++ uses 433 tokens (words, operators, delimiters, etc). Shannon uses 68 tokens. The Shannon expressions usually have the additional advantage of providing the result of an expression as a possible argument to an encompassing expression. The original Sequel2 code uses 84 tokens. That code is omitted as less interesting because Sequel2 is a special purpose language. Other significant comparisons would be with Java, Ada, Cobol, etc. All of the expressions are written in an informal, use-once style.

Shannon **C++**

Expression 1. Names of employees in Dept. 50

Shannon	C++

name of employee of dept(50)	for (cc = 0; cc < num_employees; cc++)
	if (employee[cc].dept == 50)
	cout << employee[cc].name << endl;

Expression 2. All the different department numbers in the Employee table.

dept_no of employee condense	for (cc = 0; cc < num_depts; cc++) dept[cc].flag = 1;
	for (cc = 0; cc < num_employees; cc++)
	    if (dept[employee[cc].dept].flag)
	      { cout << employee[cc].dept << endl;
	          dept[employee[cc].dept].flag = 0; }

Expression 3. Names of employees in Depts. 25, 47 and 53.

name of employee of every dept where 25 or 47 or 53	for (cc = 0; cc < num_employees; cc++)
	  if (employee[cc].dept == 25 || employee[cc].dept == 47 ||
	    employee[cc].dept == 53)
	          cout << employee[cc].name << endl;

Expression 4. Names of employees who work for departments in Evanston.

name of employee of dept of Evanston	for (cc = 0 ; cc < num_employees; cc++)
	    if (department[employee[cc].dept].location ==
	"Evanston")
	          cout << employee[cc].name << endl;

Expression 5. List the employee number, name and salary of employees in Dept. 50, in order of employee number.

for employee of dept(50) minfirst empno
          show(empno, name, salary)
	prev_num = 0;
	while 1
	{ next_num = big_number;
	    for (cc = 0; cc < num_employees; cc++)
	      if (employee[cc].dept == 50 &&
	          prev_num < employee[cc].number &&
	              employee[cc].number < next_num)
	          { next_num = employee[cc].number;
	            cc_out = cc; }
	    if (next_num == big_number) break;
	    cout << next_num << employee[cc_out].name <<
	      employee[cc_out].salary << endl;
	    prev_num = next_num; }

Expression 6. Average salary of clerks.

average (salary of clerk)	for (cc = 0, sum = 0, num_clerks = 0; cc < num_employees;
	          cc++)
	    if (employee[cc].title == "clerk")
	    { sum += employee[cc].salary; num_clerks++; }
	cout << sum / num_clerks << endl;

Expression 7. Number of different jobs held by employees in Dept.50

count job of employee of dept(50) condense	for (cc = 0; cc < num_jobs; cc++) job[cc].flag = 1;)
	for (cc = 0, job_counter = 0; cc < num_employees; cc++)
	if (employee[cc].dept == 50 && job[cc].flag)
	{ job_counter++;
	job[cc].flag = 0; }

References

[1] E. S. Lowry, “Inexcusable Complexity”, Unpublished, users.rcn.com/eslowry/inexcus.pdf .

[2] E. S. Lowry, “PROSE Specification”, IBM Poughkeepsie Laboratory Technical Report TR 00.2902, Nov 1977.

[3] E. S. Lowry, "Formal Language as a Medium for Technical Education", Proceedings of ED-MEDIA 96, p407, AACE, June 1996.

[4] M. Seltzer, “Beyond Relational Databases”, Commun. ACM 51, 7 (July 2008) 52-58.

[5]. D. Kelly, “Innovative Standards for Innovative Hardware”, COMPUTER, July 2008, 88-89.

[6] R. Stansifer, “ML Primer”, Prentice Hall 1992, 110-115.

[7] “Perspectives in Computer Science”. ACM Computing Surveys, Vol 28 No 1, March 1996.

[8] F. P. Brooks Jr, "No Silver Bullet: Essence and Accidents in Software Engineering", IEEE Computer, Vol 20 No 4, April 1987.

[9] E. S. Lowry, "Accurate description of system structure - A new standard of language quality", Proceedings of the Digital Equipment Computer Users Society, Vol 5, No2, 1978, p833.

[10] M. J. Miller. "NT 5.0 and Beyond", PC Magazine Nov 17, 1998, pg 4.

[11] E. S. Lowry, "Toward Perfect Information Microstructures", Jan. 2010, Unpublished, See users.rcn.com/eslowry/tpim.htm .

[12] D. D. Chamberlain et al, "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control", IBM J. Res. Development, Nov. 1976, pg 560.