



Fermi National Accelerator Laboratory

FERMILAB-Conf-91/270

High Energy Physics Experiment Triggers and the Trustworthiness of Software

T. Nash

*Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510*

October 1991

* Lectures presented at *The CERN School of Computing*, Ystad, Sweden, August 27-29, 1991.



Operated by Universities Research Association Inc. under contract with the United States Department of Energy

High Energy Physics Experiment Triggers and the Trustworthiness of Software

Lectures at The CERN School of Computing
Ystad, Sweden
August 27-29, 1991

Thomas Nash
Fermi National Accelerator Laboratory
Batavia, IL 60510 USA

I. Introduction

For all the time and frustration that high energy physicists expend interacting with computers, it is surprising that more attention is not paid to the critical role computers play in the science. With large, expensive colliding beam experiments now dependent on complex programs working at startup, questions of reliability -- the trustworthiness of software -- need to be addressed. This issue is most acute in triggers, used to select data to record -- and data to discard -- in the real time environment of an experiment. High level triggers are built on codes that now exceed 2 million source lines -- and for the first time experiments are truly dependent on them. This dependency will increase at the accelerators planned for the new millennium (SSC and LHC), where cost and other pressures will reduce tolerance for first run problems, and the high luminosities will make this on-line data selection essential. A sense of this incipient crisis motivated the unusual juxtaposition of topics in these lectures.

II. What's the Big Deal About Software?

Computers are fundamentally different from other technology used by society in two basic ways: the degree of system complexity and the level of human involvement. Measured in terms of the numbers and dynamics of their internal states, computers are systems of extreme complexity. This complexity carries some of the apparent opaqueness and unpredictability that is characteristic of the human personality. The sometime resemblance to human capabilities is what research in artificial intelligence (AI) tries to exploit. Computer technology is the highest of high tech. Yet, unlike other technology, it is extremely difficult to quantify. What do you mean by complexity when you don't know all the system states? What do you mean by reliability when you don't know all (and, probably, not even most) of the failure modes?

Also unique is the degree of intimate human involvement with computer systems. The human component is obvious and essential for specification, programming, definition of input, interpretation of output, and attempts to confound the system. Humans are involved in some of these activities in other enterprises, such as the development of experiment detectors, bridges, and vehicles, but never with anywhere

near the intensity required for computers. What is different is that computers aid the intellectual rather than mechanical functions of humans. These are attempts to replace our most complex and "human" capabilities. Their specification and design require a deep and most intrusive self-analysis of what we are trying to accomplish. The intensive human involvement is a profound and often neglected aspect of the computer problem.

Computer systems must be considered as part of the overall system in which they are embedded, not as one "of a collection of components" or as "appliqué".¹ In particular, it is essential that they be understood as human-computer systems. When something goes wrong, it is a failure in "integrating a complex, highly contingent world of analog variations with both human subjectivity *and* discrete-state machines."²

Computer *hardware* is subject in many, but not all, ways to the standard techniques of reliability engineering.³ Redundancy has been exploited to produce extremely reliable duplex processor machines for situations requiring high availability, such as banking, airline reservations, and telephone switching. Telephone switch downtime, planned and unplanned, is less than two minutes per year in practice.⁴ The requirement on the development, sponsored by the Federal Aviation Administration, of an automated flight-control system, including its emergency mode, is for less than three seconds down time per year. Most important, it is possible to measure hardware reliability in terms of mean time to failure (MTTF) and to predict MTTF from measurements on components for a system in which the overall complexity is kept under control. This process is what allows one to design for improved reliability with redundancy, component selection, and reduction of component count.

In computer *software* the situation is entirely different. Hardware performance and reliability have improved in the last 25 years by factors like a million and a thousand, respectively. There has been barely an order of magnitude improvement in any performance or reliability aspect of software – despite extensive research directed at what is ominously referred to as *The Software Problem*. Why is this the case? We use computers because they handle massive amounts of data that they can manipulate with data-dependent paths through complex logic. Convoluting the data states with program paths leads to a huge number of internal and final states. No human being can specify all the states for any but the most restricted applications. No human programmer can conceptualize all the states. No human tester can test all the states.

Why can't we test software until it can be certified bug-free? Because, as E. W. Dijkstra said, "Testing can show the presence of bugs, never their absence."⁵ At the beginning of his book on software testing, Glenford Myers provides a simple homework assignment for his readers.⁶ The idea is to point out how difficult it is to test fully software written to even the simplest of specifications. The spec: *read 3 integer values which are the lengths of a triangle's sides; print a message stating whether the triangle is scalene, isosceles, or equilateral*. What test cases would *you* define for a program written to this spec? Experience with real bugs for this example suggest that the test cases must satisfy at least 14 issues. (Did you check the case (0,0,0)?) A group of highly experienced programmers averaged only 7.8 out of 14. This problem is so simple that formal logic

methods could easily be used to prove the program carries out its specified task. The code required is a few tens of source lines. For large problems involving several million source lines of code (megaSLOCs), formal methods are impossible, and even highly experienced programmers would be able to define tests that would catch only the tiniest fraction of possible bugs.

Everything said here about software applies to firmware. Firmware involves data, program instructions, or logic that is fixed in the memory or the wiring layer of an integrated circuit. The name firmware applies to read-only memory (ROM), programmable logic arrays (PLAs), gate arrays, and a large variety of other devices complex enough to contain extensive information. Generally, this information is prepared in a variant software language. What goes into one chip may consist of thousands of lines of code. The problem of firmware may be even more insidious than the problem of conventional software. Firmware and hierarchical design languages (HDLs) are so intensively used in designing digital systems like computers that increasingly one sees hardware systems whose functionality traces almost entirely to software programs burned into silicon.

Redundancy works for computer hardware and for non computer systems; why not use it for software? The problem is what do you make redundant? Running two identical copies of software (on reliable hardware) will give the same result, right or wrong. The failures in software are those of the humans in this aspect of the human-computer system. So what about redundant human programmers? The idea is referred to as n -version software development.⁷ Several independent and isolated teams build software to the same specifications. The results are compared. If the number of versions, n , is greater than 2, there can be a vote. The software can even be run in separate computers. Multiple-version software and multiple, isolated, independent testers are used in critical situations to improve reliability, despite the extra expense. One of the five computers on each of the space shuttles runs a second version of the software. (In fact, a delay in the first shuttle launch was caused by a veto from this computer. The veto resulted from a synchronization bug that had 1 chance in 60 of occurring.)

Without question, this technique improves reliability, but much less than one would expect from reliability theory for systems other than software. In hardware, if the probability of failure of one component is p , the probability that a system of n independent (parallel) components will fail is p^n . If p is small, even $n=2$ will result in very high reliability. However, the redundant components in software are human programmers or testers, and they are not independent. Human errors are correlated by culture (humans read the same textbooks and learn the same algorithms) and by genetics (the thought processes in different brains may differ, but not all that much). n -version techniques are not a panacea that will eliminate software errors. As an aside, they can be useful in controlling hostile infiltration of software development groups, an issue hopefully not relevant in high energy physics, but nonetheless instructive. If there are n , independent, geographically isolated teams developing the same software, the probability of infiltrating all of them successfully does go as p^n .

Measuring software reliability is also a serious problem. Normally, to get a measure of MTTF, one averages the failure rate in a large sample. But the concept of multiple "samples" of software has no meaning. IBM has developed techniques that get around this problem for certain software products. Their answer is to measure a large sample of different usage. IBM measures the rate at which errors are found during an extensive debugging process involving release of the software to an ever-increasing circle of users. This technique is very effective for the errors of concern to IBM, which is "interested in failure-free execution intervals, rather than ... [an] estimate [of] errors remaining" IBM finds that 33% of errors are found after 5,000 user-years of being exercised (which IBM defines as 5,000-year MTTF) and therefore can be caught in a short time. The goal of these measurements is to determine a product-release date.⁸

What is special about the IBM experience is that it deals with compilers, linkers, operating systems, and other products that have a huge and diverse circulation. Users exercise all aspects of the software and quickly find the problems that most other users will find; these are the problems that matter to IBM's overall reputation. In other situations, the concern extends to the sum total of all the uncommon errors, those that could, for example, result in the crash of an airplane -- or a fatal systematic error in an experiment's trigger system. In fact, so-called fly-by-wire software can at best be measured with present techniques to have a MTTF of 10^{3-4} years. This compares to the normal standard for commercial aircraft components in the neighborhood of 10^6 years.

III. Is HEP Software Different?

What is special about experiment software compared to the IBM experience is the small number of users from a specialized, highly computer literate community who are generally quick to learn the capabilities of software and accustomed to a short time frame when responding to problems. The small number of highly similar, very sophisticated usages of a program in high energy physics tends to test the farthest corners of program space at some point in a product lifetime, but generally not early in the product life as in the IBM case. Software projects are interconnected communal efforts with many individual perspectives, software styles, and methodologies, and with varying receptivity to discipline. Multiple institution management results in limited management controls over the communal project. Furthermore, in a surprising similarity to military software, critical testing of HEP software and its use are essentially simultaneous. The actual use determines the true requirements. Implementation must respond to changes in understanding on the battlefield of experiment data runs. Repeated use of the software takes place in an environment where users, technical concerns, and understanding of goals are steadily changing. Requirements must, therefore, evolve, and iterative development of software is natural, appropriate, and inevitable.

Computing is essential to HEP experiments at four key stages: triggers, data acquisition, reconstruction, and analysis. Detectors consist of often huge assemblies of

electronic detectors that signal the presence and time of passage of ionizing particles. In many cases detector pulse height or area also indicate the amount of charge left by a track. The detector analog information is quickly converted to digital form. There is generally too much data to record, and experiment *triggers* are used to select events of interest for off-line analysis. The process of assembling and calibrating raw data from different parts of the detector and recording it on permanent storage media is known as *data acquisition*. The data is normally assembled into "events" that correspond to a single primary particle interaction. The raw digitizer data must be transformed into physics variables (momentum, mass, and originating vertex of each particle). The pattern recognition and regression procedures are known as *data reconstruction*. The resulting data summary tapes (DSTs) contain, in principle, all that was measured about the physics variables of detected particles in typically billions of interactions. The DSTs are subjected to *data analysis* by physicist written software that searches and measures new physics using simple statistical visualization displays and sophisticated fits.

IV. Multiple Levels of Triggering

The triggering of high energy physics experiments proceeds in a succession of steps that each reduce the data that must be recorded. One can identify four typical levels of triggers in the more complex large hadron collider experiments. Figure 1 shows the present conceptual data flow and triggers plans of the Solenoid Detector Collaboration (SDC) in preparation for the first SSC experiment. This structure is also seen on a simplified scale in smaller experiments and those dealing with lower data rates at electron machines.

At Level 0, fast analog electronics are typically designed to identify anything that implies an interaction has occurred. This may be an *OR* of many detector element signals or an indication that there was a beam crossing or a tagged beam particle. This trigger takes tens of nanoseconds and is used to gate front end electronics whose incoming signals had been cable-delayed to cover the decision time. It also starts the Level 1 trigger.

Level 1 triggers use the earliest available subdetector data. This data may be still in analog form, or it may have been digitized by a fast analog to digital converter (FADC), or a mixture of both. The trigger might require, for example, a significant amount of E_t or a μ track. This is intended to reduce the rate into the Level 2 trigger. In fact, Level 1 triggers are often integrated as the first step of a Level 2 trigger. These are very special purpose processors, often implemented in ECL, and generally programmed by changing parameters in a DRAM or firmware. Their time scale is about 1 - 2 μ sec.

The Level 2 trigger occurs on a time scale of 20 - 50 μ sec. It decides whether to permit a full digitization and readout of all detector signals, a time-consuming and dedicated activity that, if triggered too often, will increase the dead time of the experiment. This trigger level uses as much subdetector data as is available from FADCs and builds on Level 1 information. It often includes calorimeter cluster finding and some track

finding. Typically a processor which can be (wire or micro) programmed is used here. In the past, DSP and bit slice technology, as well as data driven processors, were commonly applied. These are generally not high level language programmable devices. The sophisticated techniques required for programming them limits access to a very small group of skilled people, effectively hiding and protecting the details from most of the members of a large experiment collaboration. Collaborations commonly interact through a trigger language or table that permits making complicated trigger choices without having to meddle with the detailed software.

At Level 3 all data has been digitized and is available, making possible triggering on a global reconstruction of data. This trigger permits recording on tape and takes 100s or 1000s of msec per event to decide. Parallel farms assembled from now generally off the shelf RISC workstation technology computers, and programmed with MSLOCs of FORTRAN, are used in an environment that is compatible with the off-line computing of the experiment. Most of the software may be almost indistinguishable from off-line. In fact, successful large program Level 3 triggers to date have depended on off-line reconstruction codes that had been thoroughly seasoned by use on a previous run's data.

The perceived availability of so much "free" computing results in Level 3 processing often being applied to data formatting and event building. This is the only real difference from off-line reconstruction and is a sign of a new trust in these systems by collaborations. Previously, specialized event builder processors were used to assemble data from across the detector that corresponded to a single interaction event, and then to pass it to the Level 3 trigger. The D0 experiment at Fermilab established a new direction by sending subdetector data on multiple data cables directly to dual port memories in their highest level trigger. A switch approach like this will also be used at the Collider Detector at Fermilab (CDF) and the SDC. CDF is using a commercial network switch (Ultranet), further demonstrating the trend to off the shelf hardware at the highest levels of data acquisition and trigger systems. Although D0 is planning to attach co-processors (Level 2-like special processors) to its commercially available micro-computers as accelerators, collaborations have clearly moved toward commercial systems that have a setting as similar as possible to the off-line.

There remains one big difference with the off-line, and this is the matter of trustworthiness in a context where data is being discarded without means of recovery. This is a critical subject to which we will return after a brief digression on the place of special purpose hardware in the on-line HEP scheme of things.

V. Special Purpose Hardware

Triggers have traditionally been seen as an opportunity for advanced and creative application of special purpose computers. Bit slice, data driven, systolic, associative string, and digital signal processors are all technologies that have been applied, or at least evangelized as appropriate.⁹ None are off-line compatible. They all require large

amounts of microcode, hardwired programming or other talent intensive effort. Ostensibly they have been used because of speed and timing requirements. Perhaps, they have also been a way to hide critical processes from prying hands behind a wall of esoteric technology. Even at Level 2, it is now possible to use high level programmable processors efficiently in a data gathering architecture. An example is the transputer trigger/DAQ system being used for ZEUS at HERA in Germany. Future switch based parallel processor standards such as the Scalable Coherent Interface (SCI, IEEE 1596) will make it straightforward to use commercial high performance, high level language programmable processors such as RISC in these applications.

So why use special purpose processors? Are they faster than RISC? A special purpose processor clocking at 50 MHz with one instruction per clock is not significantly more efficient than a streamlined RISC processor executing one instruction per 50 MHz clock. The component economics of digital processor architectures, esoteric or not, are basically the same: the number of transistors required for an instruction operating on a unit of data is almost a constant of nature. The speedup potential of incompatible special processors no longer seem compelling. Their use as subroutine coprocessors now seems even less compelling since here Amdahl's law applies: if f is the fraction of time spent in code to be run in the coprocessor, the speedup for an infinitely fast coprocessor is only $s = \frac{1}{1-f}$. If f is as high as .75, $s < 4$. In a trigger, events are discarded as you go so one can focus the speedup on early stages, but as we just noted, there isn't much, if any, speedup available from special hardware.

The exception to this poor speedup prognosis for special purpose technology is when one can train the electronics in advance and look up answers in real time. Neural networks and memory look-ups (MLUs) are two approaches that permit this. Neural networks are analog networks with weights that are allowed to relax to a state corresponding to an answer that depends on an input data pattern. They are trained by setting weights in advance by feedback (in the HEP context) with real or Monte Carlo data. Their strength is pattern recognition and they are being applied to triggers and reconstruction where it is desired to identify B physics events or separate gluon from quark jets. MLUs are large memory tables which are trained by loading results (computed in large FORTRAN programs) for all inputs (addresses). To look up an answer, one addresses the memory with the input variable and the table delivers the stored computed function of the variable as output. MLUs are particularly appropriate for quickly computing complicated functions.

Neural networks were covered extensively by other lecturers at this summer school. For MLUs the speedup potential is (almost) unlimited since in principle you can compute forever to load them up. Memory technologies have improved so much since the introduction of MLUs¹⁰ that their practicality has crossed a new usefulness threshold at 32 bits. A single bit MLU function (*to trigger or not to trigger*) of a 32 bit variable in 100 nsec is now possible for about \$35,000 with 135 SIMMs fitting on a Fastbus sized board. By mid decade this should be down to \$2000, allowing a full 32 bit function of a 32 bit variable for \$65,000 on 8 VME cards. Loading these big MLUs from a

disk image will take about a minute per output bit at 10 MBytes/sec. 16 Gbytes of disk will be required for the full 32 bit function. The practical maximum time that one can imagine for computing such a disk image is a 100 processor farm for a day. Assuming mid-decade processors, and an unoptimized loop, the corresponding speedup is at least 2×10^4 . So maybe MLUs have a future again. Since FORTRAN programs can be used to program them, MLUs are unique among special purpose systems in that they can be programmed in an environment compatible with off-line software.

VI. Trustworthiness and High Energy Physics Software

The trustworthiness of something, to paraphrase the excellent definition by David Parnas, is our evaluation of the probability that it will not cause something terrible to happen. What factors encourage high energy physicists, under the difficult circumstances of their experiments, to trust software dependent systems? How are subjective perceptions of trustworthiness developed? We will approach this business like question in a business school manner, by looking at some case studies, in the context of triggers at Fermilab's collider experiments, and drawing a set of conclusions, somewhat tongue-in-cheek and with no intent to offend.

In the past, although many experiments boasted high level triggers, few depended on them. The upcoming collider run represents the first time that major hadron collider experiments consider Level 3 farms to be a necessity. At the anticipated luminosities CDF needs a full factor of 10 reduction in its Level 3 compared to ~ 2 last run, and D0 is counting on a factor of 200. So, our first possible conclusion: *Dependence breeds confidence?*

CDF's Level 3 was used in several previous runs.¹¹ After serious difficulties in porting 2 MSLOCs of rather unstructured off-line code to the inadequate environment of the first generation of Advanced Computer Program (ACP) MC68020 based farms, the system ultimately worked and was trusted to a $\sim 2x$ cut. Once it stabilized, the Level 3 hardware and software worked well enough so that it made only a small contribution to the experiment's logged down time in the 1989 run. "This may explain the experiment's trust now," a physicist responsible in this area said. "Everyone is asking to put more into Level 3." For the 1992 run, the off-line software is now mature and trusted off-line modules will be used in the trigger. The new Level 3 trigger will be based on a SGI Powerserver RISC farm, which has a development environment that will also be used for off-line Unix reconstruction farm processing. So a second possible conclusion: *Familiarity breeds confidence?*

At D0 the equivalent of what we have defined as a Level 3 trigger is known as "level 2". It is a farm of DEC microVAX computers which corresponds to the dominant computer flavor used by this collaboration for its on and off-line activities. D0 is pushing very hard to catch up with CDF in this coming run and cannot afford an "engineering run". Therefore, they do not have the luxury of mature, run-tested off-line software to apply to their trigger. A set of special filter programs is being developed.

There is a push by those responsible to establish a "Filter Certification Board" in order to convince the collaboration that the high-level trigger is ready. This board would formally review the independent physics testing of triggers already in place. This is to preempt any temptation to use the level 3 only loosely since some might argue that a Top discovery could most safely be obtained in that way. Computer upgrades and the coprocessors are arriving late enough in order to make a few collaborators nervous, though in fact there should be an adequate 6 months to stabilize this hardware. So, two more possible conclusions from this case study: *Time pressure breeds confidence? Hardware reliability influences trust in software?*

CDF's Level 1 - 2 triggers are another interesting case.¹² They are highly trusted by the collaboration and have been for some time. The parameters for gain calibration are entered via digital tables and DACs. The collaboration incorporates its trigger decisions in a "trigger language" which is very readable and is, in principle, validatable by careful checking. Key functions are structured into μ coded routines called by the trigger language. The hardware and microcoding is the restricted domain of a few experts, which automatically puts it under tight change control and results in its being highly structured, well defined, and testable. Our possible conclusion here: *Esoteria limits access (to experts) and forces structure and project discipline far more than in open high level programming environments?*

In some ways D0 low level triggers (referred to as level 0 and 1)¹³ are similar, but because of the intrinsic simplicity of their liquid argon detector which directly measures charge, calibration is easier. Charge is injected to calibrate amplifiers and the gain, and trigonometry, is hardwired into fixed resistors which are selected and installed subject to standard, careful, quality assurance. There is some flexibility for corrections and other purposes in PROMs which are, of course, programmable. The use of firmware is very effective change control. "Burn 'em and forget 'em," was the philosophy expressed. Just as at CDF the low level trigger functions are the responsibility of a small skilled group. The collaboration's trigger choices, overseen by a Trigger Board, are incorporated in "COOR" a coordinating on line process. So our last two possible conclusions: *Geometric and calibration parameters are subject to routine QA? Reduced complexity increases trustworthiness?*

I chose to use the two Fermilab collider experiments as study subjects for obvious reasons of convenience in gathering specifics. However, it should be clear that the "possible conclusions" we identified in these case studies about how HEP experimenters establish trustworthiness would be the same with CERN experiments as examples, or experiments from any other HEP laboratory. The way in which we establish trustworthiness in our experiments is subjective and depends on an intensive and careful study of the consistency of data. Compared to what goes on outside HEP, this bottom line approach actually looks pretty good. The question we turn to now is whether it is good enough.

VII. A Crisis in HEP Software? How Do You Measure Complexity?

Some of us who inhabit the no-man's land between high energy physics and computer science, reacting probably more on aesthetic grounds than anything else to the state of HEP computing, have made hysterical sounding statements about a "crisis in HEP software". Is there a crisis?

Yes: Our detectors and collaborations are getting bigger and more complex.

No: We are doing sort of OK, intuitively structuring our problems. And it ain't broke now, so let's not risk fixing it.

I think I agree that things are not all that bad right now, but to understand what might change for the next generation of experiments, we need to understand a few things about complexity and about how we manage the software in HEP which deals with this complexity.

The first question is how do you measure complexity? The computer software industry makes a big effort to measure complexity. Accurate cost estimation is a critical factor in the profit equation, and it depends on an early understanding of complexity in a project. If you estimate a job wrong and underbid, it can be very costly. Extensive complexity estimating methodologies exist,¹⁴ based on approaches like Function Point Analysis.¹⁵ Such techniques look in great detail at the specifications for a job, accounting for what has to be done, module by module, and making reference to extensive experience-based tables -- almost like an actuary defining risk for insurance purposes. The methodologies and the experience are aimed at large software projects under tight management control. No work has been done on understanding how to estimate costs in large research-oriented software projects. A study in this direction would seem to be motivated by the number of pending big scientific projects, anointed and unanointed "grand-challenges", that depend on major software efforts.

In the absence of appropriate complexity-estimating techniques, we can take a physicists style look at big-picture quantities: the number of detector types requiring separate software efforts (we really should count "regions" -- CDF has ~100) and the number of collaborators (individuals/institutions). The detector channel count does not add to software complexity although it, of course, does affect production computing resource requirements. Here are estimates for three US hadron collider experiments:

	detector types	collaborating individuals	institutions
CDF	15	332	28
D0	9	307	31
SDC	9+	~700+	~90+

This suggests that detector complexity is not qualitatively changing. If all that matters is detector complexity, software may not get much worse at SSC/LHC.¹⁶ (However, a detailed scrutiny of the impact of individual subdetector complexity on software has not been done here -- nor has it been done by collaborations, and it is not being done for SDC. This level of complexity may be the straw that breaks the camel's back. A comparison of the CDF startup experience with D0 will be instructive, since it has been argued that D0 is a less complex more homogeneous detector.)

A part of the problem is known to scale with the number of software workers. The individual institution (university) group size seems to stay constant. Software team sizes are unlikely to change because of small group psychology issues. By a Parkinson's law kind of argument, we can expect that the resulting increase in number of software teams will result in more different things being done (more physics, more triggers, etc.) Each team will continue to interact with about the same number of teams so the total interaction complexity would seem to scale linearly with the number of people. That in itself is not a troublesome conclusion. However, Stu Loken at LBL points out that there will be an increase in the number of groups interacting with key core groups -- those with the biggest responsibilities. The complexity of key subsystems, like trigger supervisors and production managers, will be increased and their reliability may be impacted.

VIII. A Crisis in HEP Software? How is HEP Software Managed?

The other question we need to address in evaluating the extent of any future software crisis is how, in fact, is HEP software managed? First of all there is an intuitive structuring of activities. Subdetector modularity is carried into software, with hardware developers often doing "their own" software. The physics modularity is carried into software. Physics topics groups generally are responsible for overseeing topical modules of trigger and analysis software. Also as we have seen, the most critical triggers are typically handled in hardware and microcode which forces structuring and information hiding in an apparently natural and painless way -- at a considerable cost in talent.

Skeleton packages are developed by core groups and are used both for on-line trigger software and for off-line reconstruction and analysis. Physicist programs are plugged into these skeletons which provide data banks, manage batch production or trigger, sequence software subsystems, and move data from and back to tape. We will discuss later the software engineering concepts of information hiding and objects. Information hiding, which could go a long way toward improving productivity in these situations, does not appear to be applied significantly yet. The struggle to develop and maintain these packages is painful and difficulties in converting from one platform to another are symptomatic of problems that information hiding cures.

Master librarians have the role on experiments of approving inclusion of software for production and triggers. Some are assisted by review boards. The system works well if

the individual is strong and has the charisma and/or authority to impose some discipline. It is hard to identify such qualities in advance.

Perhaps the most distinguishing feature of the HEP style in software is that the fundamental approach to testing is through physics results. *The* requirement is: get the physics right. Most physicists do not know the meaning of "requirement" in the context of software or projects, and there is no other requirements document. The only accepted critical test of software is to look at results with real data: to see if trigger rates are reasonable, if event distributions are reasonable, and if anything at all looks fishy. Before there is real data, similar kind of testing is effectively done with Monte Carlo and cosmic ray data. However there is no systematic software testing, as that is commonly understood, at the component, subsystem, or system level. Furthermore, the testing tends to be schizophrenic self-testing. Independent testing by separate testers is rare. Independent testing from requirements documents does not exist. That concept is foreign to HEP.

Finally and rather unfortunately, the impact of detector complexity on software is still not a design consideration for the next generation detectors. The operative mind set, "Software has always been free. Why should we pay for it now?" was expressed at a senior level on one experiment. The real question is: is it free? And if it were cheaper, could physicists do more physics?

IX. A Crisis in HEP Software? What Has Changed.

If we look to the future, one can only anticipate that pressures will increase for large detectors to work reliably on their first time up. The history at LEP and CDF was similar: the biggest problems were in the systems with highest complexity, data acquisition, triggers, and on-line software. All of the experiments had sufficient off-line software so they were able to address "straightforward" analysis (W, Z) quickly (using an "express line" analysis at CDF) and get key first run results out. More complex areas (like B physics at CDF) were deferred to later. None of these experiments depended on high level triggers in their early running. CDF will be dependent on Level 3 this coming run, but now they have the confidence allowed by relatively stable and mature off-line software. At D0, as we have noted, it appears that the high level trigger may be considered critical in the first run. The D0 experience will be important in projecting toward SSC/LHC. SDC will likely depend on a high level trigger in their first run. It is not clear whether there will be available a straightforward analysis opportunity (like W and Z) for the first run that will promise definite results. The cost levels are high, and so will be the resulting visibility from the public, congress, and government oversight. Results will be expected quickly and one cannot anticipate much tolerance for first-run problems.

To summarize the present situation with HEP software, it is clear that how we deal with software is, at the least, unaesthetic to cognoscenti, and that the issues are difficult to identify and verbalize. Although the complexity of whole detectors is increasing only

moderately, the impact of subdetector complexity (the region count as Bob Kephart puts it) on software is not yet a strongly recognized hardware design consideration. Collaboration sociological complexity will increase somewhat, but the impact will be felt most by critical core software groups. Collaboration trust is based on indirect criteria. Approaches to software quality/trustworthiness management are currently based on very talent intensive activities which weigh hard on physicists, particularly those who might otherwise be applying their talents to more physics productivity. None of this is really new.

What has changed qualitatively, it appears, is the dependence on high level triggers. Unlike the off-line, when a trigger fails you cannot rerun data tapes into it. Data, time, and money have been lost. With the new budget scale of experiments, equipment, and operations approaching a billion dollars, we cannot afford first-run failures. The field of high energy physics will be more dependent on a few experiments. The expectations are too high and the experiments too critical to the survival of the science. All this will be amplified by the increased level of governmental oversight which expects more professional management. It is hard to believe, looking at the record in the defense sector, that this oversight will not, sometime soon, be extended to software, which will correctly be seen as critical to the success of the taxpayers' huge investment. These are all changes that require a new scale of trust in software. This is exactly in the spirit of Parnas' definition of trustworthiness: we have to reduce the probability for software inducing something terrible to happen.

X. Traditional Approaches to Engineering the "Software Problem"

The "software problem" perhaps may be understood best in terms of software's not being amenable to the usual kinds of quantitative-design disciplines practiced by engineers dealing with other technologies. The problem persists in spite of being recognized and attacked. Since as early as 1960, software engineering¹⁷ has addressed the issues of reliability in human programming of computers. In 1972 Frederick Brooks wrote the classic book on this subject based on his experience managing IBM's OS/360 development.¹⁸ It is amazing how little has been added by others in the years since.

Traditionally, efforts have focused on means of organizing programs and making them understandable and readable. During the 1960s IBM provided free-flow charting templates to programmers. It is still considered undisciplined to fail to use such tools (or modern equivalents) to prepare a diagram of the logic flow of a program prior to starting on it. The flow chart should show what happens to data, from when it is input to when it is written out, including conditional actions and branches in the program that depend on the immediate state of the data.

In the 1970s a number of different methodologies evolved and were promulgated. These formed frameworks intended to define paths toward analyzing requirements and then writing programs that are well structured and, thereby, more readable and understandable, less error-prone, and easier to test and debug. Now often generically

lumped under the terminology *structured analysis, structured design* (SASD), these methodologies are much used. In the 1980s they have been incorporated into computer-workstation software as computer-aided software engineering (CASE) tools. CASE systems are now marketed by over a hundred, mostly start-up, companies. The crusade for more structure in software has left its impact on programming languages with "structured" constructs like **do while** and **if then**. Programmers have been steered strongly away from using conditional and absolute **go to ...** branches, which, according to the dogma, represent the evil that results in much unstructured, disorganized code. SASD and CASE tools have been valuable and have improved productivity, especially for complex programs, but they have not proven to be a panacea as some had hoped. The fact is not surprising if viewed in the historical context. One experienced software engineer comments that SASD and CASE are "just glorified flow charting, no different from the 60s."

CASE tools have been used on high energy physics experiments with some limited success. These have included SASD tools as well as a HEP developed entity relation system called ADAMO which takes data entities from the screen down into the data structures of the code. The problem has been with the lack of a complete integrated CASE package. Paolo Palazzi, the developer of ADAMO, commented "... most toolkits abandon you after design. ADAMO stays with you ... but supports only one kind of abstraction..." For real acceptance in the research community, these tools must become more complete and manifestly productivity enhancing.

Another traditional feature of research focus for software engineers has been in the area of large project management. Projects are divided up into phases of requirement, specification, design, coding, and testing, very much as in other fields of technology.¹⁹ Diverse measurements support the common belief that the cost of errors increases by something like an order of magnitude for each project phase that passes before they are corrected.²⁰ Extensive project-management methodologies are in place at IBM, AT&T, and other major software and computer-system developers. These insure that there is a careful review and documentation at each phase in an attempt to catch problems early. Design and specification changes are carefully tracked and controlled because of their possible impact on other parts of a system. When the emphasis is on the quality of reviews, rather than on paperwork and bureaucracy, such project management is very effective, perhaps the most effective means to ensure production of reliable software that does what is wanted.

The Department of Defense has recognized the importance of such project management. It promulgates software standards like DoD 2167A which defines a rigorous and complex sequence of reviews and documents for DoD software contractors.²¹ DoD 2167A requirements are so pervasive in the area of defense software that companies with CASE tools for the aerospace sector advertise "automatic 2167A document generation." Contractors staff offices just to produce the requisite 2167A paperwork. A defense contractor executive with extensive project experience put it this way: "Contractors set up two teams, designers and book writers, to meet the requirements [of 2167A]. It is coincidental if the results of the two teams are the

same.”²² Military officers (like physicists) do not pay enough attention to the very difficult requirements phase, and give-and-take between those defining requirements and those specifying and designing a system is rarely adequate.

Although 2167A allows changes, it is normally used in one pass from requirements through design. The process is very difficult and results in requirements that bear little resemblance to reality. Requirements are typically 5 to 10 pages long per thousand lines of code, though there are many cases well outside this range in both directions. They are generally neither read nor reviewed properly, certainly not when the code “is over a million lines in scope”. The project can’t be stopped if, at any stage of the review sequence, it proves unsatisfactory. The executive continued: “Economic forces say proceed. So pretty soon you are reviewing something different. The books are irrelevant. You eventually have to go through a very painful merge because of the final tests.” The result is a bottom up, rather than an intended top down design. The 2167A “process fosters problems by being too detailed.” The result is that irrelevant issues get in the way of understanding what the project really needs to be able to accomplish.

XI. Modern Approaches to Software Engineering

CASE tools and the way in which DoD 2167A or other rigorous review methodologies are conventionally applied represent the traditional approach to software engineering. More than this is clearly needed. Recognition has grown widely in the last decade of two concepts that are key to further improvement in software reliability: *evolution of requirements* and *software reuse*. A realistic and complete understanding of requirements is central to the goal of software that is reliable, effective, timely, and economically affordable. Big, complex projects are just too difficult to define completely from the beginning. In the research situation, successful evolution of requirements can be accomplished by keeping documents of requirements short at first with general descriptions of what is desired. As a design proceeds, the client and designers must work together on evolving and customizing requirements. Such a process requires a commitment of time and competent personnel to a continuing requirements process. Nothing is more important for obtaining software that has a chance of being truly trustworthy during a run.

The difficulty is focusing the attention of clients, scientific or otherwise, on the requirements process. This problem is exacerbated by frequent rotation of personnel. Those with the initiative to start a project are often not around to see it through to completion and thus to influence a consistent evolution of requirements. Nonetheless, clients are rightly most concerned about their interaction with the computer system. This point of interaction is the human interface, where users input requests and receive output as display screens or paper reports. The technique of *rapid prototyping* is proving very effective in allowing quick turnaround of ideas in the development of human interfaces. Special or general-purpose computers are set up so that displays are easily changed. No consideration about performance or computer cycles is expended during prototyping.

The process of rapid prototyping involves clients deeply in the decision-making process for requirements. A quick creative process to determine what a system will be like, using (often glitzy) color displays, seems to attract and hold clients' attention at this critical phase much better than the traditional exchange of arcane documents. This fun approach to requirements focuses on displays and screens, exactly where one should focus to get started on a proper evolutionary top-down design. Unfortunately, the benefits of rapid prototyping and extensive end user involvement, are less applicable in some areas, such as in the guts of reconstruction and trigger decision programs, where there is little need for interfaces to humans during operation.

The deep involvement of users in the requirements and design process is so important to the ultimate success of a system that creative management methods are being applied to encourage such involvement. The problem comes in breaking away key client personnel from their ongoing responsibilities, the very responsibilities that the new computer system is supposed to assist. In a very successful and interesting experiment, DuPont isolated teams of users and software developers for periods of as long as six months. Each team was given a basic project definition and a "time box" in which it was to complete the new software project. The teams were to be rewarded individually and together and celebrated in the house newsletter, if they succeeded within the allotted time. All teams in a trial succeeded, and the savings in cost to the corporation, compared to estimates based on experience with traditional approaches, was extraordinary. On average, the seven initial projects came in at 33 percent of the estimates. The key to this strategy was the use of rapid prototyping techniques, combined with full-time access by software developers to knowledgeable and competent users. The user-clients' understanding of their own requirements, and the realities of implementing them, deepened during the development process. In this way, not only were systems completed with great efficiency in the use of personnel, but the systems were also more likely to meet the real needs of the organization.²³

Rapid prototyping can be made more efficient by *software storming*, a short and intensive software development effort intended to get a first order approximation of the system requirements. According to an individual familiar with this approach, "Practice has shown that the issues uncovered in this ["storm"] are the issues which require the most attention throughout the prototyping period."²⁴ Prototyping is, in some sense, testing in advance. Therefore, just like testing, prototyping is subject to missing out where scenarios are not tried. Here one never finds the last requirement, as in testing one never finds the last bug.²⁵ Involving end users in the process is not just a matter of asking them what they like. All too often they will tend towards preserving their existing work environment. Somehow in the requirements process end users must be led beyond what is familiar to new approaches that will improve their capabilities.

A rather vague concept that has received much attention in recent years is the reuse of mature (that is, relatively reliable) software. The core idea is obviously correct: to the extent you reuse reliable software, your software will be reliable.²⁶ "The most radical possible solution for constructing software is not to construct it at all."²⁷ This is exactly

how we work with PCs and Macs, we adapt canned software like Excel and Word. These are highly tested because of heavy reuse, and that motivates us to be willing to adapt our needs to their not entirely ideal capabilities. In HEP, canned software tools are used for histogramming (HBOOK, PAW), for farm and parallel computing support (CPS, Canopy), and for on-line/data acquisition (PAN-DA). These are all examples of large packages that are so well publicized it is hard not to be fully aware of them at times when they might be applied. Identifying smaller modules that might be reused is a more difficult problem.

The hype associated with software reuse mostly concerns the automatic cataloging of software and what it is meant to do. Automatic cataloging of software is an unrealistic promise that has been made by some in the artificial intelligence community. Clearly, if it were possible to define for a computer exactly what you wanted, and if there were a catalog of what existing software could do, all categorized well, then the problem would be solved. You would just click your mouse, and, voilà, the software you needed would be ready to reuse, in perfectly (and relevantly) documented form.

The idea behind reusing software is more subtle than the hype indicates. Two software-engineering approaches, one dating back to 1970, are important to the reusability goal. These are *information hiding*, a concept suggested by David Parnas in several early papers²⁸, and, what can reasonably be considered its descendant, *object-oriented programming*. Parnas's deceptively simple idea is to have small teams of coders work on software modules. These modules communicate only through extremely well-defined interfaces. The modules contain deliberately hidden information, one or more secrets that define how the module operates. Other modules – and teams – know the interface definition but need not and should not know any but their own secrets. The secrets may, for example, refer to hardware-specific matters that are in this way isolated to single modules. In normal programs a change of hardware can have consequences throughout a huge software package. Parnas's secrets restrict consequences to single modules. Since consequences are kept local, information hiding allows easy evolution of requirements and change of software. It also is effective in allowing reuse of these very well defined and consequence-isolated modules. Information hiding encourages compartmentalized, well defined tasks, which are easier to categorize for possible reuse. It does nothing to assist in the dream of automatic cataloguing. But at least, if a module is remembered, it is likely to be reusable.

Object-oriented programming has gained notice in recent years, much of it as a result of the development of the Smalltalk language by the Xerox Palo Alto Research Center (PARC).²⁹ Smalltalk was subsequently discovered by Steve Jobs on a legendary visit to Xerox PARC. It was then applied to the famous graphical human interface of Apple's Macintosh computer. Objects consist of data and one or more methods (like program procedures) that operates on the data. One communicates with objects by passing messages between them telling them what to do. The well-defined message-passing interfaces and the consequence-isolated methods of the objects are certainly within the spirit of the information hiding philosophy. A class of objects is defined once; it may have many objects that are instances of the class (such as the multiple windows on a

Macintosh). Classes, with their own specialized traits, may inherit general traits from other classes (for example, the traits of a basic window may be inherited and augmented by scrolling capabilities). One reuses what is needed and never has to describe anything twice. This is true software reuse, building on what has been done before. A large object-oriented program is built up – and tested – object by object, in an evolutionary way that clearly reduces bugs. The ease of changing objects carries with it a disadvantage from a project management perspective, the difficulty of controlling change.³⁰

Evolving requirements implies that software is not finished after its initial “build”. This has led to the concept of *incremental builds* where the ever changing nature of software is recognized from the outset. Really, as Brooks noted, programs are grown not built. This concept must be very familiar to high energy physicists whose basic approach to programming is incremental. Information hiding is exactly what one needs to support this (good) habit in an efficient way because it insists on a structure that is receptive to unanticipated changes and additions.³¹

Information hiding and object oriented programming offer the possibility of a painless discipline for HEP software that fits well the incremental build approach commonplace in the field. The experience from other areas is that these techniques add only small execution inefficiencies while providing significant productivity and reliability enhancements. Professionally developed trigger, reconstruction, or analysis skeletons, within which physicist written programs could be incorporated, are natural opportunities for applying information hiding or objects. The skeleton could reach a high level of trustworthiness through some of the traditional and modern disciplined approaches we have described. The physicist code would be caged, so to speak, in regions of asymptotic freedom where the only required discipline would be at the defined interfaces on the boundary.

Another, very different, approach to increasing trustworthiness is the use of formal methods to prove (jargon is *verify*) that the program does what the specification says.³² The process of being sure that the specification says what the humans want is known as *validation*. The specification is written in a formal top-level specification language (FTLS) which is more “English-like” than conventional languages. After conversion, the result is code about ten times longer than the FTLS spec. Formal methods are mathematical proofs that the FTLS and the computer program are equivalent. Program source code and specification languages present the same scale of difficulties in assuring their correctness. Formal methods, therefore, are essentially amplifiers by an order of magnitude of the size of assuredly correct software segments, from about 1,000 lines to about 10,000. Though this may be extended somewhat by a hierarchical application of formally proven modules, one soon gets into the problems of measurable software reliability discussed in an earlier section. A further problem is that as the size of formal mathematical proofs increases they reach a point where confidence in the correctness of the proof itself is at issue.

Formal methods were applied by the National Security Agency's National Computer Security Center (NCSC) at the highest levels of its standards for "trusted computer systems". Here a *Trusted Computing Base* is verified with formal methods. Honeywell's Secure Communications Processor (SCOMP) is an experimental system at the A1 summit of the NCSC classification. Only twenty copies of SCOMP have been installed because of its limited scope and poor performance. Despite a limited scope, the formal specification for SCOMP requires the equivalent of over twenty dense pages of mathematical spec notation. This amount must be somewhere near (some evidence suggests it is beyond) the maximum that humans can produce with reasonable assurance that it is error-free. Note that even at this A1 summit of security, there is no requirement to prove that the code implemented is equivalent to the verified design! Despite these difficulties, formal methods might have applicability in high energy physics trigger software, where one can imagine developing verified trusted trigger kernels. These would be treated like the microcoded kernels in use on Level 2 triggers and called by a trigger decision language.

When considering which other modern software technologies might be able to help in improving software, one cannot neglect artificial intelligence (AI). Because of an excessive level of historical hype and unrealizable promises made in the mid 1980s, AI has been somewhat discredited. This may be an overreaction. With moderate expectations, one important subfield of AI, expert systems, does have potential relevance to high energy physics software. Expert systems are rule-based programs for tasks requiring expertise. They tend not to use deductive logic, but rather abductive reasoning. The word *abduction* describes the unnatural process of generating explanations, cause from effect. Normal logic proceeds by deducing effect from cause. Abduction involves such heuristic techniques as plausible inference and the weighing of evidence. Most important, for lack of a broad understanding of the world shaped by common sense, expert systems must be limited to narrow and isolated domains.³³

Famous examples of the effective use of expert systems are diagnosis of breakdowns of railroad locomotives, and, in the case of Digital Equipment Corporation's VAX series of computers, determination of appropriate customer configurations. Expert systems appear to present a significant debugging problem when they get large. This implies that an inherent size limit may come into play in the future.³⁴ The process of obtaining knowledge from a human expert (on locomotives, for example) and putting it into a form that an expert system can refer to goes by the loaded name, *knowledge engineering*.

Not surprisingly getting experts to agree, when there is more than one, is a significant problem. It is an interesting, perhaps more than semantic, question whether expert systems dispense knowledge or doctrine. One can imagine that when the use of medical diagnostic systems becomes widespread, any medical doctor who ignores the questions and conclusions of the systems will risk malpractice suits. Expert systems are really storehouses of doctrine.

Their creation forces a systematic analysis of information and procedures. They can provide a framework to organize and control change in large, complex research

projects. This could be applied effectively to HEP data acquisition, control systems, and software in contexts like operation, diagnosis, repair, and updates. Diagnosis in the absence of an expert (owl shift guidance on Fastbus, for example) is the classic application on experiments. One can imagine extending this to software maintenance situations such as porting code to new platforms and system releases. An expert system could be used for acceptance tests self-administered by physicists before their new code is certified for inclusion in production packages. Although expert systems are reasonably effective at disbursing expert information and doctrine, loading them with the information is not straightforward. An effective means for do-it-yourself knowledge engineering would make it possible to broaden the application of expert systems in high energy physics - - and many other fields.

I have intended, in this discussion, to give a flavor of what is unique about “the software problem” and why intuition and experience gained from other technology areas can be very misleading. The emphasis here has been on describing how the problem is being attacked. This is one good way of viewing this rather intangible subject and sensing how difficult the struggle to manage software development really is. Looked at from another direction, when computer systems are successfully applied, as they so often are, this serves to whet the appetite for applying them to more complex problems. A prime cause of the software problem is that our ability to imagine ever more ambitious systems is not limited. What is limited is our ability to understand and express, and then to produce and test, what we imagine, precisely enough to make our dreams work properly in reality.

Most important is to understand the extreme difficulty of making big advances in the productivity of writing software or the reliability of the result. There are no simple solutions, “no silver bullet” as Brooks titled a recent paper.³⁵ One needs to exercise considerable judgment in walking the knife edge between anarchy and the potential for sloppy unreliability, on one side, and excessively bureaucratic philosophy constraining creativity and changeability on the other. The trustworthiness of HEP software will relate directly to the sensible use of techniques and methodologies of modern software engineering.³⁶

XII. Conclusion: Elements of an Advanced Software Engineering R&D Agenda

Several possible R&D directions in software engineering have suggested themselves as the discussion in these lectures developed. They included

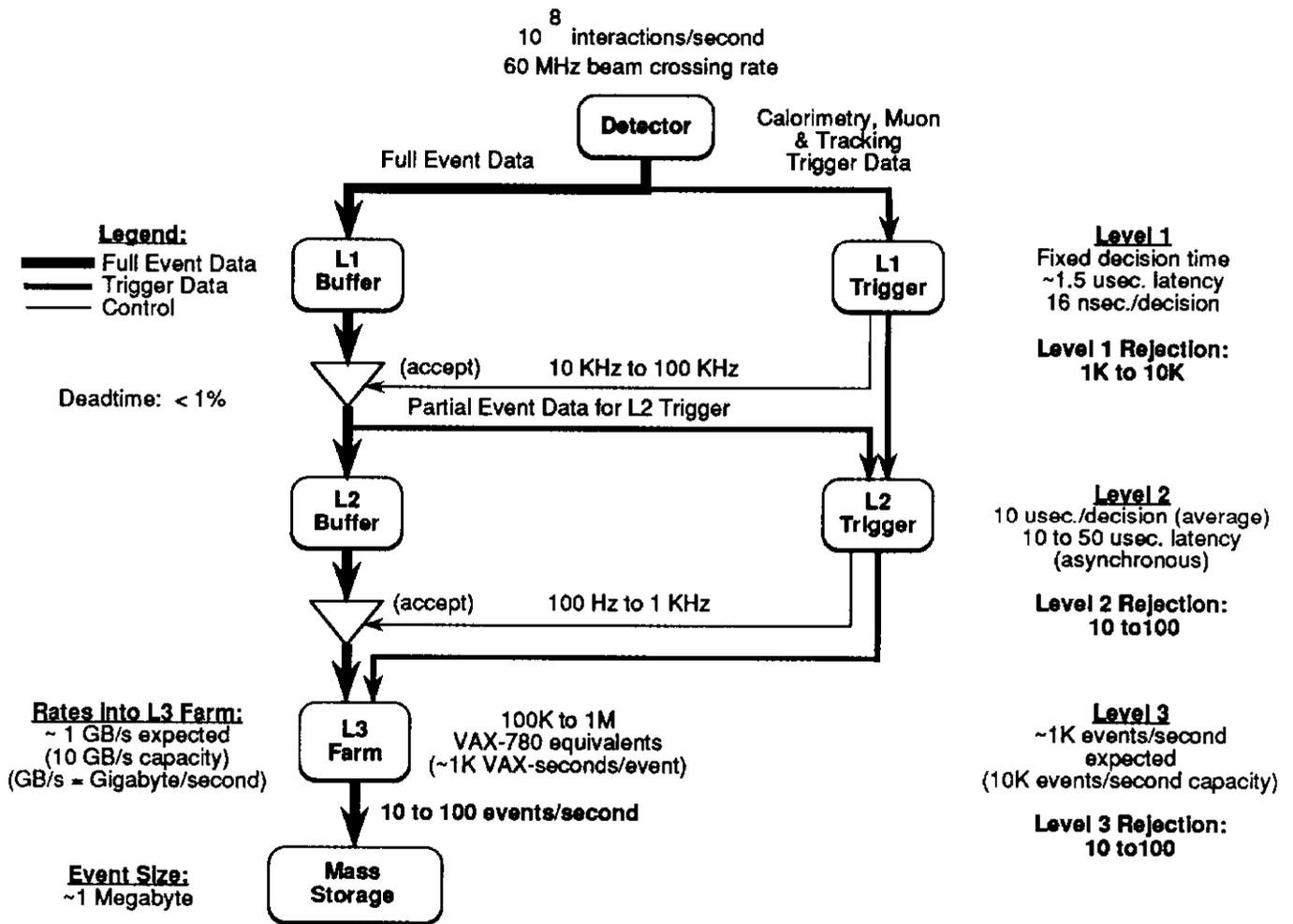
- Complexity estimation methods for large research projects that take into account the impact of hardware design and the sociology of research groups.
- Software project management methodologies for big science research projects in undisciplined multi-institution environments. Workable review and independent testing methodologies appropriate in these conditions.

- Integrated CASE tools for big research projects that incorporate project management and review and testing requirements.
- Painless structuring techniques for large scientific projects that apply information hiding and/or object oriented software techniques.
- Tools for easy entry of expert knowledge and doctrine into an expert system. Do-it-yourself knowledge engineering.
- Application of formal methods to create trusted trigger kernels.

Many of these problems require a study of group dynamics and software management in a large project research environment. Most, if not all, such work was previously in the context of business, computer industry, and (to a lesser extent) military software projects. As has been the case in other technology areas, it is likely that the true software engineering needs of HEP, once we look at them closely, will push the technology envelope in advance of the needs of the rest of society. These problems appear to attract considerable interest from research software engineers and organizational behavior social scientists. The result of such work will not be a silver bullet for HEP's -- or anyone else's -- software problem. And physicists do not recognize a software crisis at this point. Nonetheless, even small improvements in scientific productivity and the reliability of large project software are likely to be critical to the continued success and acceptance of large budget, multi-year scientific efforts. Serious consideration must be given to high energy physics driven efforts in advanced software engineering research.

Acknowledgements

I would like to thank Jim Patrick, Liz Buckley, Marvin Johnson, Jim Linnemann, Maris Abolins, Gene Fisk, Stu Loken, Irwin Gaines, Luann O'Boyle, and Nelly Stanfield for recent discussions and special help with these lectures and my colleagues in the Fermilab Computing Division for stimulation over the years. This paper draws in part on work carried out while I was on sabbatical at the Center for International Security and Arms Control at Stanford University and uses some material published in a working paper by the Center.³⁷ I was supported at Stanford in part by the Carnegie Foundation through a Carnegie Science Fellowship. Fermilab is funded by the Department of Energy.



SDC Event Flow Diagram

Figure 1. Conceptual data flow and triggers for the Solenoid Detector Collaboration (SDC) system at the Superconducting Super Collider (SSC).

References

- 1 This point was emphasized in the SDI context in Report of the Defense Science Board; Task Force Subgroup on Strategic Air Defense (SDI Milestone Panel), Office of the Under Secretary of Defense for Acquisition, Washington, D.C. , (1988), 3. The Eastport Study Group, "Summer Study 1985," in A Report to the Director of Strategic Defense Initiative Organization (Washington, D.C., 1985.)
- 2 Gary Chapman (President, Computer Professionals for Social Responsibility), personal communication, August 10, 1989.
- 3 Daniel Siewiorek and Robert Swarz, *The Theory and Practice of Reliable System Design*, (Bedford, Mass.: Digital Press, 1982).
- 4 W. N. Toy and L. E. Gallaher, "Overview and Architecture of the 2B20D Processor," *The 3B20D Processor and DMERT Operating System*, (Bell Laboratories, Naperville, IL) No. 2, (April 1983), 8.
- 5 J.Dahl, E.W. Dijkstra, and C.A.R. Hoare. "Notes on Structured Programming," *Structured Programming* (London: Academic Press, 1972), 6.
- 6 Glenford J. Myers, *The Art of Software Testing* (New York: Wiley, 1979), 1.
- 7 J. C. Knight and P. E. Amman, "Issues Influencing the Use of N-Version Programming," (unpublished paper, Software Productivity Consortium, Reston, Va., 1988).
- 8 Harlan Mills *et al.*, *IEEE Transactions on Software Engineering*, SE12,(1986), 3.
- 9 Irwin Gaines and Thomas Nash, *Use of New Computer Technologies in High Energy Physics*, Ann. Rev. Nucl. & Part. Sci. 37, 177 (1987).
- 10 E. Barsotti *et al.*, Trans. Nucl. Sci. 26(1), 686 (1979).
- 11 J.T. Carroll *et al.*, NIM A 300, 552 (1991).
- 12 D. Amidei *et al.*, NIM A269, 51 (1988); G. Ascoli *et al.*, NIM A269, 63 (1988); G.W.Foster *et al.*, NIM A 269, 93 (1988).
- 13 M. Abolins *et al.*, NIM A289, 543 (1990).
- 14 Barry Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
- 15 International Function Point Users Group, *Function Point Counting Practices Manual, Rel. 3.0*, April 1990, Westerville, OH 43081-4899.
- 16 P. Kunz, Proceedings Computing in HEP, Oxford, 1988.
- 17 At this mention of "software engineering", it is important to remind that "engineering" here does not carry with it the usual implication that we are referring to a quantitative discipline, like, say, "mechanical engineering". We do not know how much stress will cause a software structure to collapse. In fact, as I try to show here, we do not even know what such a question means in any quantitative sense.
- 18 Frederick Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering* (Reading, MA.: Addison Wesley,1972).
- 19 One should note, as David Weiss comments, the significant absence of a production phase in software technology.
- 20 Barry Boehm, *Software Engineering* [IEEE Trans. Comp. C -25 (1976)].

-
- 21 Department of Defense, *Military Standard: Defense System Software Development*, DOD-STD-2167A (Washington, D.C., 1988).
 - 22 Private communication 1989.
 - 23 Peter R. Mimno, at seminar, "CASE Tools, Comparison and Review," CASEXpo, Anaheim, CA, 1988.
 - 24 Andrea Weiss, Mitre Corp., private communication, August, 1989.
 - 25 Frank Maginnis, Mitre Corp., private communication, August 3, 1989.
 - 26 See, for example, *Proceedings, Workshop on Reusability in Programming*, (Newport, RI: ITT Programming, 1983); and Will Tracz, ed., *Software Reuse: Emerging Technology* (Washington D. C.: IEEE Computer Society Press, 1988).
 - 27 Frederick Brooks, *No Silver Bullet - Essence and Accidents of Software Engineering*, *Computer*, April, 1987, 10.
 - 28 D.L. Parnas, "A Technique for Software Module Specification with Examples," *Communications of the Association for Computing Machinery (ACM)* 15 (1972): 330-36, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. of ACM* 15 (1972):1053-58, and "On the Design and Development of Program Families," *IEEE Transactions Software Engineering* SE-2 (1976):1-9.
 - 29 A good introduction is Mark Stefik and Daniel Bobrow, "Object-Oriented Programming: Themes and Variations," *AI Magazine*, 6, 4 (winter 1986): 40-62.
 - 30 For more on object oriented programming, see Paul Kunz's lectures at this summer school.
 - 31 David L. Parnas, *Designing Software for Ease of Extension and Contraction*, *IEEE Trans. Softw. Eng.* SE5, 128 (1979).
 - 32 Daniel Ince, *Software Development: Fashioning the Baroque* (Oxford: 1988); *An Introduction to Discrete Mathematics and Formal System Specification* [Clarendon, 1988].
 - 33 More details on AI and expert systems may be found in the lectures by K.H.Becks at this summer school.
 - 34 David Weiss, private communication, August, 1989.
 - 35 Frederick Brooks, *No Silver Bullet - Essence and Accidents of Software Engineering*, *Computer*, April, 1987, 10.
 - 36 Gene Layman and J. R. Robins, "Open Architecture for Modern Command and Control Systems," *Signal*, (August 1988): 83-89.
 - 37 Thomas Nash, *Human-Computer Systems in the Military Context*, (Stanford, CA: Center for International Security and Arms Control, 1990).

Figure Caption

Figure 1. Conceptual data flow and triggers for the Solenoid Detector Collaboration (SDC) system at the Superconducting Super Collider (SSC).