Software Engineering Research Laboratory: A White Paper

Nancy Leveson

Aeronautics and Astronautics
Massachusetts Institute of Technology
leveson@mit.edu

Computers are rapidly becoming an integral part of nearly every engineered product, as well as controlling the manufacturing process for products: Computers control consumer products, commercial aircraft, nuclear power plants, medical devices, weapon systems, aerospace systems, automobiles, public transportation systems, and so on. Virtually nothing is engineered and manufactured in the U.S. today without computers affecting the design, manufacturing and operation. Not only do products use computers to operate better or cheaper—"smart" automobiles and appliances are examples—but complex systems are incorporating designs that cannot be operated without computers—for example, unstable aircraft and space vehicles that cannot be operated successfully by humans alone. David Hughes wrote in a recent editorial in Aviation Week and Space Technology:

> "Information technology is becoming a key part of everything the aerospace and defense industry does for a living, and as the century closes it is computers and software that hold the keys to the future. The [aerospace] industry is being transformed from dependence on traditional manufacturing into something that looks more like IBM and Microsoft with wings."

At the same time that computers are becoming indispensable in controlling complex engineered systems, quality and confidence issues are increasing in importance. We are hearing more and more about failures due to computers: Software errors have resulted in loss of life, destruction of property, failure of businesses, and environmental harm. Computers now have the potential for destabilizing our financial system. Some large government-financed projects are in trouble or have been canceled because of difficulty in assuring the quality of the software.

One of the reasons for the problems is that these systems require that standard engineering techniques be extended to deal with new levels of complexity, new types of failure modes, and new types of problems arising in the interactions between components. Computers exacerbate engineering problems by allowing levels of complexity and coupling with more integrated, multiloop control in systems containing large numbers of dynamically interacting components. We are attempting to build systems where the interactions between components cannot be thoroughly planned, understood, anticipated, or guarded against. The fundamental problem is intellectual unmanageability: Increased complexity and coupling make it difficult for the designers to consider

all the potential system states or for operators to handle all normal and abnormal situations and disturbances safely and effectively. The failures in these systems are arising in the interactions between components. While we train engineers to be experts in individual fields, these complex heterogeneous systems (composed of electromechanical, digital, and human components) require knowledge and techniques that span engineering disciplines.

The Software Engineering Research Laboratory is a response to these problems. It's goal is to act as a focus for interdisciplinary research, education, and development to support the engineering and use of computers embedded in and controlling complex engineered systems. This white paper discusses the problem being attacked, attempts to delineate why the problems have not already been solved, and suggests some specific research topics that we feel are of critical importance in stretching the current limits of complex system engineering.

1 The Problem

During and after World War II, technology expanded rapidly, and engineers were faced with designing and building more complex systems than had previously been attempted. The creation of systems engineering as a discipline received much of its impetus from aerospace programs, but the new systems engineering techniques were soon adopted and applied to the process industry (chemicals and nuclear power), transportation systems, and other complex engineered systems.

As the systems we wanted to build became too complex or too time-critical to be controlled by humans or even electromechanical devices, computers started to be used to take over at least part and sometimes all of the control functions. Not only are computers flexible and seemingly limitless in their power, but they work at a speed that cannot be duplicated by any other means and are relatively cheap besides. These characteristics allow us to engineer products and complex systems that were previously inconceivable. The computer has freed us from many of the physical limits of electromechanical devices, but we are now faced with practical limitations in our ability to engineer the software parts of these systems.

As electromechanical controllers are replaced by computers, many of the basic engineering and systems engineering techniques that were developed to cope with complex systems are no longer adequate. Software adds the potential for introducing a level of complexity not previously possible: Most control software is too complex for complete mathematical analysis and yet too structured for statistical analysis. At first, heroic human effort, brute force techniques, and tremendous amounts of money were able to get large software projects like the Space Shuttle control system finished successfully. However, our ambitions are starting to outstretch the limits of what brute force and money can accomplish, and the technology to build such systems and to provide the needed confidence in their quality does not exist.

As an example, the Space Shuttle software, one of the largest and most ambitious software development projects of the 1970's, contains about 400,000 lines of code. NASA put enormous amounts of money into its development and still spends approximately $100,000,000 a year to maintain it. In contrast, even automobiles and some household products now have or will soon have that much software in them. More complex projects, such as upgrades to the U.S. Air Traffic Control System, Space Station Freedom, commercial and military aircraft, and even telephone switching systems contain millions of lines of code. To build such software may require hundreds and sometimes thousands of people, and just organizing these projects is a massive undertaking. The result of not solving these system and software engineering problems may be failures in our

attempts to build the complex systems of the future. As just one example, the huge cost overruns and technical difficulties encountered in building a new U.S. Air Traffic Control system led to cancelling large parts of it a few years ago. The more recent scaled back attempts to provide limited upgrades are also running into problems. The past six months have seen the failure of five satellite launch attempts, several of them blamed on software, including the most recent failure of a Titan IV-B/Centaur Milstar mission that has been billed as the most costly unmanned accident in the 50-year history of Cape Canaveral launch operations.

Merely producing enormous amounts of code is not enough. The potential for losses—human, environmental, and financial—with these computer-controlled systems makes quality of paramount importance. Virtually all non-trivial software has errors in it, and we do not currently have the capability to locate and correct these errors. We are putting reliance on human products that we cannot demonstrate are trustworthy, and it is getting worse as the complexity of the systems we attempt to build increases.

While the U.S. has been ahead of the rest of the world in software engineering, this situation is starting to change. The EEC countries and the Japanese are catching up and may be ahead in achieving high quality levels. Currently, the Japanese outstrip the U.S. in quality and productivity for relatively simple software systems, and they are now working on the engineering of more complex systems. The EEC countries have launched major initiatives in software engineering, including applying mathematical techniques to software, and are now ahead of the U.S. in this and other areas. The center of gravity of software engineering research in general may now have shifted to Europe.

2 Why the Problems

Although major initiatives are currently missing, certainly a great deal of effort has been and still is being applied to these problems. Why are we still having trouble building embedded software?

One answer to this question is that we have made progress, but the problems we are facing are increasing at a faster rate. The term "software crisis" to describe the problems of software engineering was introduced in the late 1960s and still is being used. However, this usage is misleading. Today we have relatively few problems building the typical software systems of the 1960s. Man's reach always outdistances his grasp—as we learn how to build one type of software system successfully, we immediately want to accomplish more.

But we cannot blame all our limitations on increasing expectations. Although a large number of researchers have been working on software engineering, their results have had limited use in real systems. There may be several reasons for this.

First, academic researchers have concentrated on the mathematical aspects of problems and solutions while ignoring human factors and the necessarily informal aspects of software development. While mathematical techniques are useful in some parts of the process, informal techniques will always be a large part (if not the majority) of any software development effort, and, indeed, most engineering projects in general. Researchers often focus exclusively on formal or on informal aspects of software development without considering their interaction.

Formalism is crucial in developing software for critical systems, but the limits of modeling reality must be taken into account: (a) the actual system has properties beyond the model, and (b) mathematical methods cannot handle all aspects of system development. No comprehensive approach to developing critical systems will, in the foreseeable future, be entirely formal

while informal approaches alone cannot provide adequate confidence. Our approaches must be driven by the need to systematically and realistically balance and integrate mathematical and nonmathematical aspects of software development.

Often the result of research is methodologies that cannot be incorporated into practice by developers and maintainers. Developing understanding about how to build critical software is not enough. The methodologies must include training and technology transfer and must be usable by those with typical software engineering backgrounds. The methodologies must also incorporate models that are closely related to the problem domain and the way that application experts think about their problems, not necessarily the way that researchers look at the problems.

One serious drawback of past and current software engineering research is lack of scalability. Researchers have developed techniques that work only on small systems. Mathematical techniques have, for the most part, been used only on very limited properties and on unrealistically small problems. Most any analysis technique works on a toy problem. There is reason to believe that software development in the large is so different than the toy problems found in most research papers that many published techniques may not apply to real projects. We need to find a balance of formal and informal techniques that scale by considering, from the start, problems of realistic size and complexity. Software engineering researchers rarely validate their techniques and theories on realistic software. Given the complexity of the systems we are attempting to build, the *only* convincing argument that an approach will work in practice is to validate techniques on real systems.

Successfully building software for complex systems demands that qualities such as reliability, safety, security, and timing be rigorously addressed and systematically built into the software from the beginning. In addition, simply concentrating on initial development is not enough: These qualities must be preserved as the software evolves during its lifetime. Independent efforts to ensure individual qualities in narrow domains, e.g., security, have made significant progress. However, no approach exists that combines diverse techniques into an *integrated* methodology for developing and maintaining software for critical systems. Furthermore, the methodologies that are developed must be usable by other than their developers and must be able to be incorporated into practice by software developers.

3 Specific Areas for Research

We believe the following areas are of special importance and difficulty in engineering complex, computer-controlled systems and thus are appropriate avenues of research. Many of these research goals are at the interface of what has typically been considered software engineering concerns and those of system engineering.

3.1 Modeling and Analysis

Whereas in the past engineers were able to reuse standard designs that had been perfected over many years, most of the new systems using computer control require new designs. The complexity of these systems, furthermore, does not usually allow us to build physical prototypes and experiment with them enough to learn how to improve our designs. Instead, mathematical models must be used to verify certain required properties. An important research topic involves defining powerful and efficient modeling languages and analysis techniques to allow prediction

and accumulation of information that will aid in the system and software design and verification process. Although many modeling techniques have been proposed, most consider only very limited system aspects and do not adequately handle such things as timing, failures, and hazards.

Analysis is an intrinsic part of any engineering discipline—no bridge or space vehicle is constructed without enormous amounts of modeling, calculation, checking, and revision. Today's software engineer simply lacks the theory to bring to bear on engineering problems. Gerhart has suggested that the scientific basis that currently exists is a collection of *micro-theories*, each reasonably well understood but isolated by its own notation, techniques, and world view. Most models are related to single qualities, such as security or reliability. A few general models exist with extensive theories, such as Petri nets, but these models often lack the power to provide the required information to designers or to address the variety of qualities required in large and complex systems. Most models also provide little help in comparing alternative system designs.

Not only do we need better formal methods, but we need ways to interface them to human abilities and to informal methods. The techniques and tools we develop must be usable by software developers and not just by the researchers that developed them, and they must be integratable into normal software development environments.

3.2 Engineering for Quality

One of the most important issues in complex systems is achieving and assuring quality—identifying and resolving tradeoffs between various qualities, determining how to achieve multiple qualities, and providing confidence or assurance that particular systems will exhibit required qualities over their lifetime.

Essential system-wide properties (reliability, safety, security, and modifiability) must be built in from the beginning; they cannot be added on or simply measured afterward. Up-front planning and changes to the development process are needed to achieve particular objectives. These changes include using notations and techniques for reasoning about system properties, constructing the system to achieve particular properties, and validating (at each step so that it is done early) that the evolving system has the desired properties. Central to this problem is the consideration of the interactions among critical system properties and potential conflicts among them. Research about different kinds of properties are usually associated with distinct, often insular, groups.

An unwarranted assumption is often made that independent approaches to achieving specific software and system qualities can be easily composed. Unfortunately, this is not true. As just one example, approaches to ensure usability or reliability properties may (and often do) interact in important but indirect ways with approaches to ensure safety properties. Many techniques can be found to attack particular subproblems, but these techniques may not be easily integrated or may be too costly if very different procedures are required for each critical property or if each part of the software development process does not build on the results obtained in the previous steps. We need integrated methodologies for developing and maintaining software that encompass the entire development process and consider multiple and perhaps conflicting goals.

3.3 Providing Assurance

More than half of software development effort goes into confidence building activities (verification and validation). We are able to execute and test only a small fraction of the possible system states

before software is put into operational use. Yet, particularly for critical systems, high confidence is often a prerequisite for certification or use.

While dynamic analysis, i.e., testing, will always have an important place in providing confidence, cost and criticality are increasing the need for static analysis of software that can provide assurance over the entire range of software states. Testing and analysis should and can support each other, with testing providing confidence in the correctness of the assumptions made in static analysis. We need to provide more affordable and effective testing while at the same time exploring the potential for static analysis of important properties and understanding the interaction between these two approaches to assurance.

3.4 Human-Computer Interaction

Most complex systems require a combination of human and computer control, where humans provide intelligence and problem-solving ability while computers handle aspects requiring speed and computational power. Challenges exist in determining how to allocate tasks between humans and computers and how to design the features of this interaction so that the unique capabilities of each are optimized. Simply replacing the human by computers, the obvious and often only approach considered, may not result in the most efficient, useful, and safe systems. The desired end is a partnership between the computer and the human that is superior to either of them working alone.

Serious accidents are starting to occur in aircraft and other shared control systems where the design of the interaction between computers and humans is being blamed rather than failures or errors on the part of either of these system components. Although much research exists on how to make usable and "friendly" computer interfaces, very little exists on how to integrate computers and humans in a complex system.

In a slightly different context, a better understanding also is needed of the way to design software engineering tools and languages in order to minimize the number of errors that are introduced during software development and to provide usable and useful tools to software developers. One of the roadblocks in making progress on these problems is the lack of scientifically established information upon which to make decisions about the design of software engineering tools and techniques. There has been a great deal of study of the mathematical and engineering foundations of software engineering, but much less of the psychological foundations. We need to establish these foundations.

3.5 Evolution

Software engineering approaches often concentrate on initial software development and not on the continual evolution of the software and its environment. Software is continually changing and evolving, not only because of the discovery of latent errors, but primarily because of changes in the operating environment, in the needs of the end users, and in the underlying technology. Software must be designed to be changeable without compromising the confidence in the properties that were initially verified. Sometimes decisions will have to be made not to change critical software if the risk is unwarranted. We need ways to make those decisions, ways to design and construct software so that it can evolve over time without compromising critical properties, and techniques to aid in the evolution and change process itself.

3.6 Risk Assurance and Assessment

Computers currently are being introduced into the control systems of dangerous processes (such as nuclear power, public transportation, and weapons) without any way to determine whether the associated risk is reduced, the same, or increased. Because analog and mechanical control systems with measurable risk are being replaced by computers, we need to develop procedures that provide the same level of assurance of acceptable risk.

Numerical risk assessments of physical systems usually are derived from (1) historical information about the reliability of individual components and models that define the connections between these components or (2) historical accident data about similar systems. Neither of these assessment approaches apply to software: Historical information is not available, software is usually specially constructed for each use, and random wearout failures are not the problem. Devising probabilistic models of software reliability is an important research topic; they are potentially very useful in software development. But their usefulness in certifying safety is less clear.

The very low failure probabilities and high confidence in these assessments that is required in safety-critical systems require more experience with the software than could possibly be obtained in any realistic development process. More important, these models are measuring the wrong thing. Software reliability is defined as compliance with the requirements specification, but accidents most often occur as a result of flawed specifications, i.e., faulty assumptions about the behavior of the environment or the required behavior of the software. Software reliability prediction models assume that it is possible to predict accurately the usage environment of the software and to anticipate and specify correctly the appropriate behavior of the software under all possible circumstances. Both of these goals are impossible to achieve.

Probabilistic evaluation may not be the most effective way to achieve confidence that software will *always* do the correct or safest thing under *all* circumstances. An emphasis on formal and informal verification, analysis, and review may be more appropriate in evaluating a software and system design. We need more research on procedures to identify software-related hazards, to eliminate and control these hazards through design, to apply safety-analysis techniques during software development to provide confidence in the safety of software and to aid in the design of hazard protection, and to evaluate the effectiveness of the analysis and design procedures to assess the level of confidence they merit.

Qualitative risk assessment and assurance techniques need to be developed if government and society are going to continue to allow the use of computers to control processes that potentially affect public safety.

4 Summary

Industry and government are currently struggling with building complex, computer-controlled systems, and often unsuccessfully as witnessed by failures of major projects. We envision the Software Engineering Research Laboratory as a place where academia, industry, and government can come together to focus on stretching the limits of the complexity of the systems we can successfully engineer.