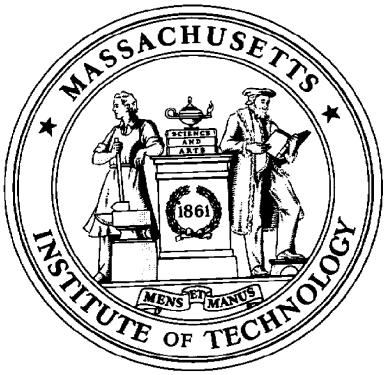


REPORT -- LEAN 95-01

Toward Lean Hardware/Software System Development: Evaluation of Selected Complex Electronic System Development Methodologies

Prepared by: Alexander C. Hou

Lean Aircraft Initiative
Center for Technology, Policy, and Industrial Development
Massachusetts Institute of Technology
77 Massachusetts Avenue
Cambridge, MA 02139

February 1995

The author acknowledges the financial support for this research made available by the Lean Aircraft Initiative at MIT sponsored jointly by the US Air Force and a group of aerospace companies. All facts, statements, opinions, and conclusions expressed herein are solely those of the author and do not in any way reflect those of the Lean Aircraft Initiative, the US Air Force, the sponsoring companies (individually or as a group), or MIT. The latter are absolved from any remaining errors or shortcomings for which the author takes full responsibility.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
Introduction

1.1 Overview

Since the first flight of a heavier-than-air aircraft at Kitty Hawk on December 17, 1903, aircraft designers have achieved tremendous performance gains in speed, altitude, payload, and range. While these improvements are impressive, evaluating the progress of aeronautics merely in terms of these performance measures would not provide an adequate description of the technology's evolution. In recent years, revolutionary developments in digital electronics and communications have altered basic concepts in military aircraft design and operation. For instance, in the past, with only mechanical flight controls at their disposal, engineers had no choice but to design inherently stable aircraft. Today, digital fly-by-wire control systems make possible the design of highly agile aircraft which are unstable—and would otherwise be uncontrollable by a pilot. The impact of information technologies on weapons systems is even more pronounced. Early air combat involved pilots from opposing sides shooting at each other with infantry arms they had carried in their cockpits. Today, engagements can take place beyond visual range and bombs can follow laser beams to their targets.

The advent of information technologies has enabled the Department of Defense (DoD) of the United States to acquire systems with capabilities far beyond what had been state-of-the-art just a few years earlier or had not even existed. Between 30 and 40 percent

of the development and procurement costs of a new weapon system can be attributed to electronic hardware and software. As systems get "smarter", this percentage will only increase. With the declining level of defense expenditures driving industry to adopt lean production practices, the development process for hardware/software systems must be a focal point of efforts to get more "bang for the buck".

The goal of lean development of hardware/software systems poses complex development and acquisition challenges for DoD and American industry. Currently, the development of hardware and software in the defense industry is complicated by the following factors:[1]

- "Material needs" can be satisfied by many combinations of mechanical and electronic systems (hardware and software).
- Technology development processes are heavily influenced by the DoD acquisition process.
- This DoD acquisition process, which evolved to procure mechanical systems, is mechanically-oriented and frequently has difficulty when developing information-based weapons systems.
- Traditional methods partition an electronic system into hardware and software elements and develop these elements separately, despite the tight coupling between "hardware" and "software" in most complex electronic system design problems.
- The administratively driven development process of defense electronic systems is often slower than the evolution of basic electronic technologies, which means that the final program result may be more costly than similar commercially available systems.

[1]These factors were identified in the course of field research conducted by Martin Anderson and Alex Hou in support of the Lean Aircraft Initiative consisting of numerous interviews with both government and industry officials. The interview sample included officials from both sponsoring and non-sponsoring companies and government agencies.

As part of the Lean Aircraft Initiative, a research program studying the applicability of lean production principles to the defense aircraft industry sponsored by the Air Force

and over 20 aerospace companies, this report evaluates a set of five complex electronic system development methodologies for applicability as a lean electronic hardware/software system development methodology and analyzes the implications of the evaluation results.[2]

1.2 Outline of Report

This report is based upon a combination of extensive literature review, including the most contemporary public documents from the Department of Defense, and upon field and phone interviews conducted under the auspices of the Lean Aircraft Initiative.

The remaining sections of this chapter discuss the challenge of reconciling the strategic needs of the American military with the declining level of defense expenditures in the post-Cold War world, the increasing importance of electronics and software in military systems, and the development challenges that have arisen as a result.

A set of new methodologies for complex electronic system development is discussed in Chapter 2. Each of these methodologies has demonstrated significant improvements in development performance or shows potential for similarly significant improvements.

Chapter 3 describes the criteria that were developed to evaluate the methodologies for possible application as a cross-system integration methodology. Each methodology is evaluated, and the results are discussed. Details of each individual evaluation are also included.

Chapter 4 discusses the implications of the evaluation results. A possible foundation for a lean hardware/software development methodology is described. Conclusions and recommendations derived from this research are summarized. The final section details recommendations for further study.

A more detailed technical description of the practices behind one of the complex electronic system development methodologies evaluated in this report is provided in the Appendix.

---

[2] This report was derived from research performed in the development of a master's thesis written by this author.

1.3 THE NEW CALCULUS

The end of the Cold War injected a high degree of uncertainty into the national security planning process of the United States. For decades, the subject of how to defeat the numerically superior forces of the Soviet Union and its Warsaw Pact allies in wartime had been the focus of defense planners, strategists, and wargamers in the U.S. and the other North Atlantic Treaty Organization (NATO) countries. Suddenly, our sworn enemies had become our new friends, triggering euphoria over the promise of a new world order and a peace dividend. The depolarization of the world left defense planners without a clear threat to replace the Soviet Union.

However, while the collapse of the Soviet Union has fundamentally altered U.S. strategy and force planning, the need for powerful and decisive U.S. military capabilities endures. If the United States is to remain engaged in world affairs, the ability to bring military power to bear when appropriate to protect its interests, as well as those of its allies, must be maintained. Although there are a wide range of potential military threats to American interests, regional conflicts have become the new focus of U.S. military planning. These types of conflicts present several challenges for the U.S. military including numerous potential locales, smaller forward deployments, short warning times distant deployments, and increasingly capable weapons in the hands of adversaries.

As a part of the reexamination of U.S. national military strategy, the Joint Chiefs of Staff (JCS) recommended that the United States should field forces capable of defeating aggressors in two concurrent, geographically separated major regional conflicts (MRCs). [3] Recently, an evaluation of the capability of U.S. forces to achieve key operational objectives in future major regional conflicts was published by RAND. This study, called *The New Calculus: Analyzing Airpower's Changing Role in Joint Theater Campaigns*, took the two-MRC requirement as a given element of national military strategy and assessed U.S. military capabilities to fulfill the mission—focusing particularly on possible means of enhancing airpower's capabilities in joint operations.

[3] In this context, concurrent major regional conflicts are conflicts that erupt sequentially but overlap so that they must be prosecuted simultaneously at times.

RAND's analysis concluded that the projected capabilities of U.S. forces would enable it to satisfy the two-MRC requirement, although the effectiveness of forces in the second theater would be highly dependent on the degree of concurrency of the two conflicts as well as the outcome of the first MRC. Regarding the role of *airpower,* it concluded that *"the calculus has changed and airpower's ability to contribute to the joint battle has increased"* (Bowie et al., 1993, p. 83). The combination of modern airpower's lethality in conventional operations, which has been greatly enhanced by the employment of advanced precision-guided munitions and modern C4I (Communications, Command, Control, Computers and Intelligence) systems, and its strategic mobility and survivability make it a good match for the needs of short-warning MRCs.

To fully exploit the potential of airpower, the RAND study made a number of recommendations aimed at ensuring that U.S. forces could establish and maintain air superiority and enhance its ability to contribute to other aspects of the joint battle. Detailed simulations indicated that equipping current fighters with AMRAAM (Advanced Medium Range Air-to-Air Missile) would ensure air superiority until some time around the year 2000. However, to ensure air superiority over the long term, simulations indicated that a next generation platform, such as the F-22, would be needed in addition to the continued development and procurement of advanced air-to-air missiles (Bowie et al., 1993).

The recommendation to equip our future air forces with more advanced munitions extended beyond the air superiority role to the strategic air offensive and ground campaigns as well. To supplement existing U.S. capabilities—based mainly on fighters and sea-launched cruise missiles—in strategic air offensive operations, the study advocated equipping long-range bombers with precision-guided munitions and standoff weapons, significantly increasing both the effectiveness of early attacks on strategic assets and the rate of destruction of these targets. To enhance the ability of U.S. forces to halt the advance of enemy ground forces and establish an assured defense, RAND's analysis indicated that employment of dispensers equipped with smart anti-armor submunitions, such as the Sensor Fuzed Weapon (SFW), could stop a force of 10 armored and mechanized divisions in approximately half the time required by the same forces armed with current weapons. Furthermore, B-2 bombers equipped with inertially-guided dispensers filled with smart submunitions could be used to provide additional anti-armor capability in the early stages of the conflict and further decrease the time required to halt an armored invasion (Bowie et al., 1993).

The analysis also indicated a need to procure additional fighters such as the F-15E, whose long range, heavy payload, and modern avionics make it a highly effective and versatile asset. Finally, a rapidly deployable theater C4I system—a goal believed to be achievable through the integration of current systems provided that planned upgrades materialize—was deemed essential to the effective and efficient prosecution of airpower's missions within the joint operations framework (Bowie et al., 1993).

Although equipping our forces with advanced munitions, advanced fighters, and rapidly deployable theater C4I systems would allow a smaller force structure to support U.S. national military strategy, these enhancements would surely require a considerable investment. Appropriating funds for this purpose could be difficult since changes in the international security and economic environments have created momentum for the downsizing of the U.S. military and decreasing levels of defense expenditures. This is perhaps the real "new calculus"—cost is now as important as system performance. With the major budgetary impact being felt in procurement which is estimated to be down 47 percent from the peak years of the buildup during the 1980's, the greatest challenge for DoD in the post-Cold War era may be how to maximize its "bang for the buck".

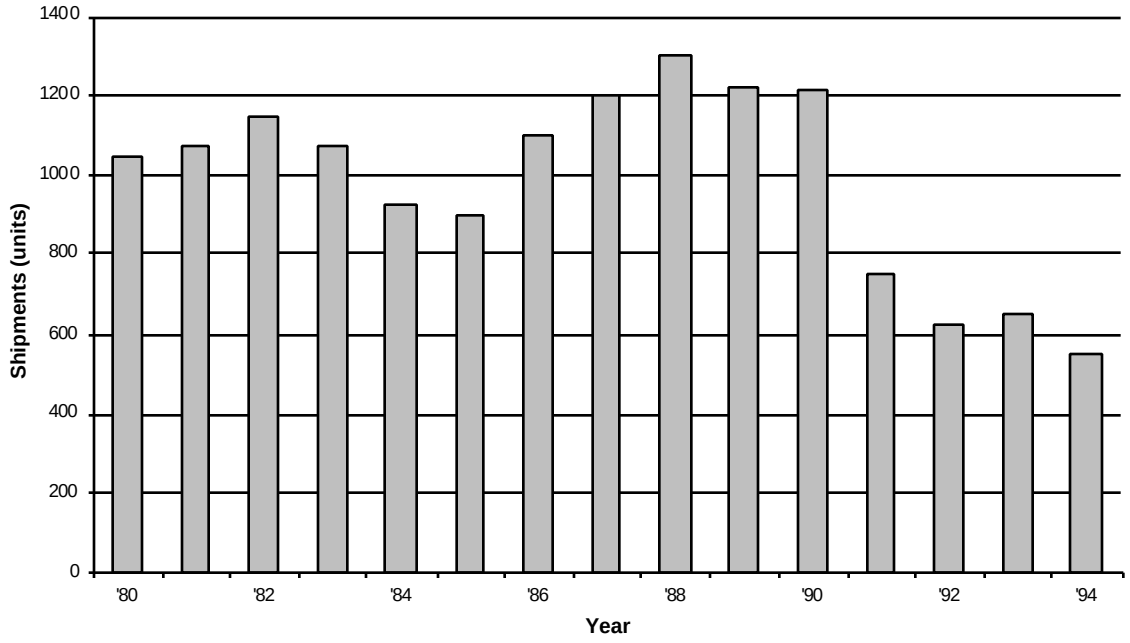
If achieving greater efficiency has become an imperative for DoD, it has become a matter of survival in the aerospace industry. Aerospace industry shipments in 1993 fell 11 percent in real terms and were also expected to fall 11 percent in 1994 from 1993 levels (DoC, 1994). Historically, the industry earned at least half of its revenues from military sales. The worldwide decline in defense spending has reduced the demand for military aircraft, missiles, avionics, and other related equipment from U.S. suppliers. The most recent DoD budget request represented a cumulative real decline in defense spending of more than 40 percent since the peak of the buildup in 1985 (DoC, 1994).

Unlike past downturns in defense spending, the commercial sector has experienced a concurrent slump in demand for its products and is unable to sustain the industry's current level of capacity. Adding to the overcapacity problem are aircraft manufacturers from the former Soviet Union—currently operating at production rates less than one-third

of capacity—who have joined the fray in vying for military aircraft sales in the export market (DoC, 1994).

While the aerospace industry in general has suffered greatly during the recent downturn, the military aircraft sector, where the U.S. Government historically accounts for 80 percent of all sales with Foreign Military Sales and direct exports collectively accounting for the remaining 20 percent, has been particularly hard hit by declining defense procurements (DoC, 1994). The resulting downward trend in total shipments of complete U.S. military aircraft is shown in Figure 1.1. While intensifying competition for shrinking defense procurement dollars has driven some companies to diversify into commercial markets or sell off their defense businesses entirely, many have decided to remain focused on the defense market and outlast the competition. For these companies, improving the efficiency and the effectiveness of their operations through the reengineering of business processes and implementation of leaner practices is paramount.

Figure 1.1: Shipments of U.S. Military Aircraft (1980-1994)



Estimates and forecasts for years 1993 and 1994 by International Trade Administration.
Source: U.S. Department of Commerce, *U.S. Industrial Outlook 1994.*

1.4 Importance of Hardware and Software

Electronic hardware and software are important elements in all the key factors for dramatically increasing U.S. capabilities for destroying enemy forces cited in *The New Calculus:* advanced munitions, avionics, and aircraft and enhanced and rapidly deployable theater C4I capabilities, such as those provided by AWACS and JSTARS (Bowie et al., 1993).

Another indicator of the importance of electronic hardware and software is the high electronic content of military systems. A chart illustrating the forecasted range of variation of electronic content for some typical defense systems is shown in Figure 1.2. Here, electronic content is the percentage of defense procurement and RDT&E (Research, Development, Test and Evaluation) outlays that are devoted to electronics hardware and software. Clearly, the data in Figure 1.2 indicate that all of the selected systems possess a substantial level of electronic content. Even aircraft, which have the lowest level of electronic content of the set of selected systems, are expected to have levels of electronic content ranging between 30 and 35 percent. In modern aircraft development programs, avionics can be the "killer" expense, costing around $7000 per pound to develop (Rich and Janos, 1994). Missiles contain a higher level of electronic content, and the theater C4I systems—ARSV (Airborne Reconnaissance, Surveillance & Verification) systems, electronics and communications systems, and space systems[4]—possess the highest electronic content levels of all. Overall, the substantial levels of electronic content indicate that electronic hardware and software are integral elements of our military capabilities.

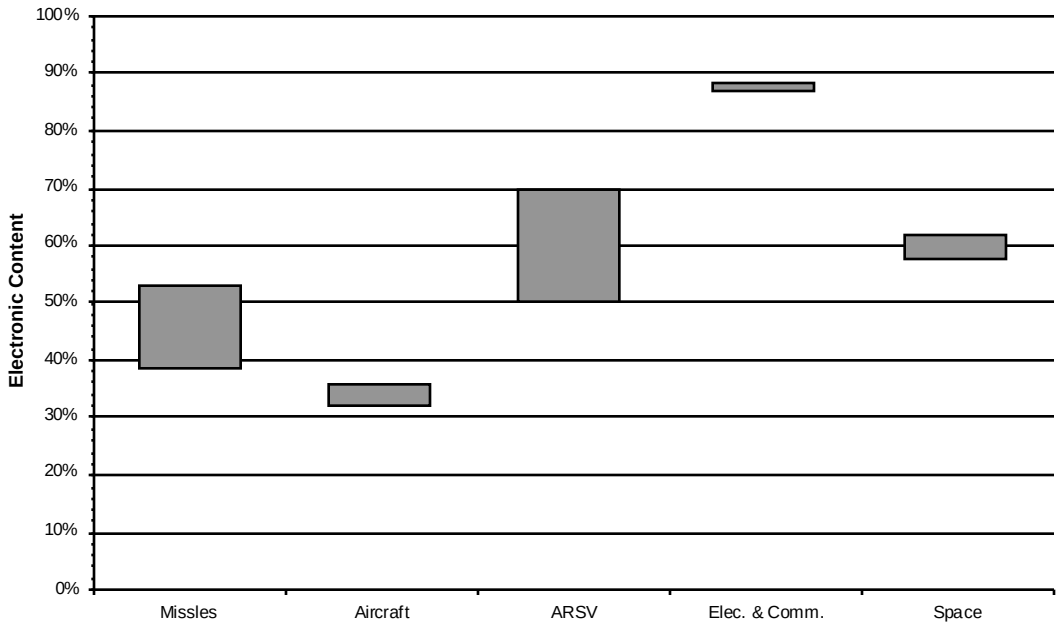
Perhaps the most convincing indicators of the importance of hardware and software are the enhanced military capabilities that are made possible by electronic systems.[5] To fully appreciate the significance of these capabilities, it is useful to consider the past. During World War II, it required 4,500 sorties by B-17 bombers and 9,000 bombs to accomplish what a F-117A can do on one sortie with a single bomb (Toffler and Toffler, 1993).

---

[4] Space systems include C4I assets such as early warning satellites and communications satellites.

[5] In fact, many believe that the widespread use and proliferation of advanced information and communication technologies is driving a revolution in warfare (Toffler and Toffler, 1993; Krepinevich, 1994; Arquilla and Ronfeldt, 1992; Hou, 1994).

Figure 1.2: Electronic Content Variation for Selected Systems, 1990-2001



Electronic content was calculated as percentage of procurement and RDT&E outlays devoted to software and electronic hardware. Forecasts for the years 1992-2001 by Electronic Industries Association.
Source: EIA (1991)

During the Vietnam War, accomplishing the same mission required 95 sorties and 190 bombs (Toffler and Toffler, 1993). While attempting to destroy the Thanh Hoa bridge, American pilots flew 800 sorties and lost ten planes without achieving success. The bridge stood intact until a flight of four F-4s armed with some of the earliest smart bombs accomplished the task in a single pass (Toffler and Toffler, 1993).

Whereas the crew of a Vietnam-era M-60 tank had to find cover and come to a stop before firing, the crew of a modern M-1 Abrams tank can fire while on the move (Toffler and Toffler, 1993). Similarly, while the chances of a M-60's crew being able to hit a target 2,000 yards away at night are slim, night-vision devices, laser ranging systems, and computerized targeting systems which compensate for heat, wind, and other conditions assure that the M-1 crew will score hits nine times out of ten (Toffler and Toffler, 1993). Clearly, the armored groups whose tanks are equipped with these enabling systems have a significant edge over forces that are not. As was the case with the early smart bombs, the Vietnam War also demonstrated the utility of advanced avionics. Before the fielding of advanced bombing systems, pilots could not do much "jinking" (erratic flight) and still

have any chance of delivering their payload on target. The advanced bombing systems compensated for altitude, speed, and a moderate amount of jinking, affording the pilot a significantly higher degree of protection while enhancing his accuracy (Momyer, 1978). Modern systems can enable a pilot to use dumb bombs with a high degree of accuracy.

During the war in the Gulf, precise position information provided by Global Positioning System (GPS) receivers to coalition forces allowed the desert sands to be navigated with a high degree of confidence (Schwarzkopf, 1992). This capability was crucial since the desert was devoid of landmarks—even the sand dunes shifted. The capability enhancing potential of GPS was clearly demonstrated in the first strike of the war by the successful helicopter raid on the Iraqi early-warning radar sites. According to the Pentagon's final report on the Gulf War, the raid was made possible because of night- and low-light vision technologies and the precise navigational capability afforded by the Global Positioning System (DoD, 1992).

While the media made the public aware of the capabilities of precision-guided weapons and other enabling technologies such as night-vision goggles and GPS receivers during the Gulf War, two of the most powerful information weapons of all—AWACS and JSTARS—were relegated to relative obscurity. The E-3 Sentry, otherwise known as AWACS (Airborne Warning and Control System), is a modified Boeing 707 aircraft, crammed with computers, radar, communications gear, and sensors. In both Desert Shield and Desert Storm, the AWACS aircraft scanned the skies in all directions to detect enemy aircraft or missiles, sending targeting data to ground units and interceptors.

The ground-scanning counterpart of AWACS was the Joint Surveillance Target and Attack Radar System (JSTARS). The E-8A JSTARS aircraft is a modified Boeing 707 equipped with a multi-mode radar for detection and tracking of enemy forces, processing equipment, mission crew work stations, and command and control interfaces. Data collected on board are then relayed to six ground station modules that receive radar data processed by the aircraft in real time. The data can then be analyzed by ground commanders for battlefield application (Swalm, 1992).

At the start of the Gulf crisis, the JSTARS system was still in development testing and at least three years remained before an initial production decision was to be made.

However, its potential for locating Iraqi tanks was so impressive that the only two existing prototypes were deployed to Saudi Arabia (Swalm, 1992).

JSTARS was initially limited in operations to performing a surveillance role. After only two days, this limitation was removed and a weapons allocation officer was assigned to control his own F-15Es, especially in the campaign against tactical ballistic missile sites. Over the course of operations, Joint STARS evolved to serve in a C4I (Command, Control, Communications, Computers and Intelligence) capacity as part of an interconnected network of these assets, which included AWACS, the RC-135 Rivet Joint electronic eavesdropping aircraft, the Airborne Command and Control Center (ABCCC), and various Army and Air Force command and intelligence centers. Linking JSTARS and AWACS together provided coalition commanders with a comprehensive picture of enemy tactical movements on the ground and in the air (Swalm, 1992).

By all accounts, JSTARS was a boon for coalition forces.[6] Ground commanders could track the movements of enemy forces on a real-time basis, from as far away as 155 miles, under all weather conditions. Aircraft directed by Joint STARS had a 90 percent success rate in finding targets on the first pass, and close air support and interdiction aircraft consistently ran out of ammunition before they ran low on fuel once JSTARS became operational (Swalm, 1992). By the end of the war, the two JSTARS aircraft had flown 49 sorties, logging 535 hours of flight time, successfully detected over 1,000 targets, and controlled 750 fighters. Attesting to the system's revolutionary capabilities, Air Force Chief of Staff General McPeak said, "We will never again want to fight a war without a Joint STARS kind of system."

---

[6] For interdiction missions, JSTARS could use its synthetic aperture radar to provide real-time damage assessment and direct immediate re-attacks. On one occasion, two A-10s and an AC-130 directed by JSTARS destroyed 58 out of 61 vehicles in convoy (Swalm, 1992). In another instance, an Iraqi unit mustering to attack VII Corps was 80 percent disabled before it could engage any of the corps's units (Swalm, 1992). In a similar scenario, a unit of the Republican Guard preparing to launch a counterattack was detected by JSTARS and targeted from a ground station and destroyed by Army Apache attack helicopters (Swalm, 1992). JSTARS also played a significant role in hunting mobile Scud launchers, first locating their positions and then passing that information on to ground-based and airborne strike assets (Swalm, 1992).

1.5 DEFINING THE DEVELOPMENT CHALLENGE

While electronic hardware/software systems are integral to the performance of defense systems, the development of hardware and software in an effective and efficient manner remains an elusive goal. High quality and reliability are extremely important since even small errors can severely degrade the performance of the system.

For instance, the AMRAAM program experienced both the difficulties involved in integrating AMRAAM software with various aircraft systems and the adverse effects of minor software problems. In one four-on-four test, all four missiles failed to hit their targets. Three missiles failed because the radar detected false targets. The failure of the fourth missile was caused by a software problem—a constant that the missile's computer used in some calculations was wrong—that required only a few lines of code to be changed (Mayer, 1993).

In addition to the challenge of developing and fielding high-quality hardware and software, it is also necessary to consider other factors that should shape the development challenge. The following is a list of some of these factors:

a) *Declining defense outlays*. Cuts in the defense budget will result in lower levels of development and procurement expenditures.

b) *Long service lives*. Historically, defense systems tend to have longer service lives than were originally intended at the outset of development. In addition, since DoD will not be able to afford as many new systems as it has in the past, fielded systems and newly developed systems may have their service lives stretched out even longer than in the past.

c) *Rapid improvement of technology.* Modern weapons systems have a high degree of electronic content. The performance of electronic hardware and software technologies continues to advance rapidly and shows no signs of slowing down. The performance of these technologies that are such an integral part of theater systems can improve dramatically over the service lifetime of a fielded system.

d) *Growth in complexity*. Demands for ever more advanced capabilities and the rapid improvement of technology have created a rapid increase in the complexity of new electronic hardware/software systems.

Factor **a** highlights the importance of ensuring the development of electronic hardware/software systems in an affordable manner. Factors **b** and **c** point to the need for newly developed electronic hardware/software systems to be upgradable, evolvable, and maintainable. Factor **d** points to the need for a development methodology which can be scaled to cope effectively with the increasing complexity of advanced electronic systems.

Thus, a lean hardware/software system development methodology must be scalable and be capable of producing a high-quality product that is affordable, upgradable, evolvable, and maintainable.

CHAPTER 2
New Methods for Complex Electronic System Development

To effectively address the challenges of developing modern complex electronic hardware/software systems, a lean development methodology needs to be devised. With this in mind, a set of complex electronic system development methodologies was investigated with the hope of finding a methodology that could be directly applied in this role or could at least provide a starting point for the development of a suitable methodology. This set included the following methodologies:

- The rapid development process used to develop the flight control software for the DC-X
- The GritTech rapid development process
- Ptolemy-supported hardware/software codesign
- The RASSP (Rapid Prototyping of Application Specific Signal Processors) design methodology[1]
- Cleanroom software engineering

Each of the methodologies described in this chapter has produced significant improvements in the development of complex electronic systems or shows great potential

[1] Note that both Ptolemy-supported hardware/software codesign and the RASSP design metodology are grouped together loosely under section 3.3, “Hardware/Software Codesign.”

for producing similarly significant improvements. These methodologies will be evaluated in Chapter 3 for their applicability as a lean hardware/software development methodology.

"Complex electronic system" is an umbrella term used to denote any system or group of systems which contain a large amount of electronic[2] hardware and software. Complex electronic systems can range from RAM-based field programmable gate arrays or digital signal processors running assembly code algorithms to JSTARS and beyond to include what we have denoted as "theater systems".

These systems are inherently "complex" because of the dramatic increase in the complexity of the hardware and software that are used in the development of even "simple" products. Consider, for instance, the development of a printed circuit module. The number of semiconductor gate equivalents contained by a typical (6 inch x 9 inch) printed circuit module increased by a factor of 20 between the 1980s and the 1990s. In the 1980s one of these modules ran at clock speeds between one and five megahertz. In the 1990s a module of similar size might run at clock speeds of 50 MHz or more. The size and complexity of the software contained in the modules has also displayed a similarly explosive rate of growth. Today, one of these modules may store more than four megabytes of code and/or data.

This growth in complexity is also evident at the avionic system level. The F-4s that saw combat in Vietnam did not have a digital computer on board and had no software. The first fighter aircraft equipped with digital computers were developed in the 1960s. These systems, which performed fire control tasks, required between 100 and 200 four- to eight-bit words of assembly language code. In the 1980s the software requirement had grown to approximately 400,000 eight- to sixteen-bit words of a combination of assembly language and higher-order languages to perform more complex functions including fire control, navigation, engine control, built-in-test, electronic warfare, flight control, stores management, and controls and displays (EIA, 1988).

This shift from solving relatively simple problems in software to solving problems of much greater complexity occurred during the mid 1970s. The F-16As, developed during the 1970s, were equipped with seven computer systems, 50 digital processors, and

[2] For the purposes of this report, other types of systems, such as photonic or optoelectronic, could also fall under the broad category of "electronic systems."

135,000 lines of code (EIA, 1988). Produced in the late 1980s and early 1990s, the F-14D has 15 computer systems, 300 digital processors, and 236,000 lines of code. The B-2 reportedly has over 200 processors and approximately 5 million lines of code (Anderson and Dorfman, 1991). Currently, the F-22 has approximately 1.3 million lines of code on board, and has 4 million lines of code when the fighter's support systems are included.[3] Even transport aircraft exhibit this explosive growth in hardware and software complexity. The C-17, which is the most computerized, software-intensive, transport aircraft ever built, has 56 computerized avionics subsystems which use 19 different types of embedded computers incorporating over 80 microprocessors and nearly 1.36 million lines of code (GAO, 1992).

The explosive growth in the complexity of hardware/software systems has made the development of these systems all the more difficult. The rate of increase in difficulty of system design and integration problems threatened to outstrip the rate of improvement of methods developed within existing paradigms. This realization provided the impetus for the creation of the methodologies discussed at length in the remaining sections of this chapter.

2.1 RAPID DEVELOPMENT DC-X STYLE[4]

For the past several years, McDonnell Douglas Aerospace-West (MDA-W) has been developing a flight software development process known as RAPIDS (Rapid Prototyping and Integrated Design System). McDonnell Douglas Aerospace has applied RAPIDS to more than 15 projects including several flight and ground test programs including Single Stage Rocket Technology (SSRT) and Ground Based Interceptor. Phase II of the SSRT Program, awarded to MDA-W in August 1991, provided another opportunity to demonstrate the effectiveness of this methodology.

As a result of customer insight, software development, cost, schedule and reliability issues were closely scrutinized early in the program. This provided MDA-W with the

[3] Telephone interview with Chris Blake, Avionics IPT leader, F-22 SPO. July 16, 1994

[4] This section is based upon phone interviews with Mr. Matt Maras of MDA-W and Dr. Jo Uhde-Lacovara of JSC's Rapid Development Laboratory and publications which they provided (Maras et al., 1994; Uhde-Lacovara et al., 1994).

opportunity to propose the use of a non-traditional process based on an integrated system level approach to the Guidance, Navigation, and Control (GN&C) design.

The entire Operational Flight Program (OFP) used by the Delta Clipper Experimental (DC-X1) was designed, coded, integrated, and tested in 24 months. The 66,000 source lines of Ada code were used by the DC-X1 to complete nine system level static fire tests and three fully successful flight tests. Encompassing an autonomous GN&C capability, the flight control portion of the OFP was developed entirely within the integrated design and rapid prototyping environment.

In addition to the flight software (FSW) developed with this approach, test code, representing high fidelity models of the vehicle, its aerodynamics, sensors and actuators, and winds were also designed by employing the same methodology. Almost 70 percent of the new vehicle software developed for the DC-X1 program was produced with the integrated design environment, using automated code generation (Maras et al., 1994).

2.1.1 The Traditional Approach

The traditional approach to developing flight software displays the symptoms of what Hammer and Champy (1993) refer to as "process fragmentation". Engineers skilled in particular problem domains formulate detailed requirements for the systems and subsystems. During the requirements design phase, engineers develop and test candidate GN&C algorithms. A non-real-time engineering simulation is created to compare the performance of different algorithms. Several reviews are scheduled during this phase resulting in the elimination of some algorithms from further consideration. Following the selection of an algorithm or a set of algorithms, a requirements document is written.

These requirements are then passed on to other organizations which interpret the requirements and translate them into actual computer code. Once the code is written and tested, it is delivered to organizations responsible for the integration of the hardware with the software and testing of the resulting system. Typically, this is where many unforeseen problems arise—late in the schedule where problems are most difficult and costly to fix. Corrections are especially costly when changes to the requirements must be made. For example, a change in the mission requirements may necessitate changing the flight software requirements. These changes can require extensive modifications to the FSW. To

make matters worse, the change process is usually conducted in the same fragmented, sequential manner as the original design iteration.

2.1.2 The RAPIDS Process

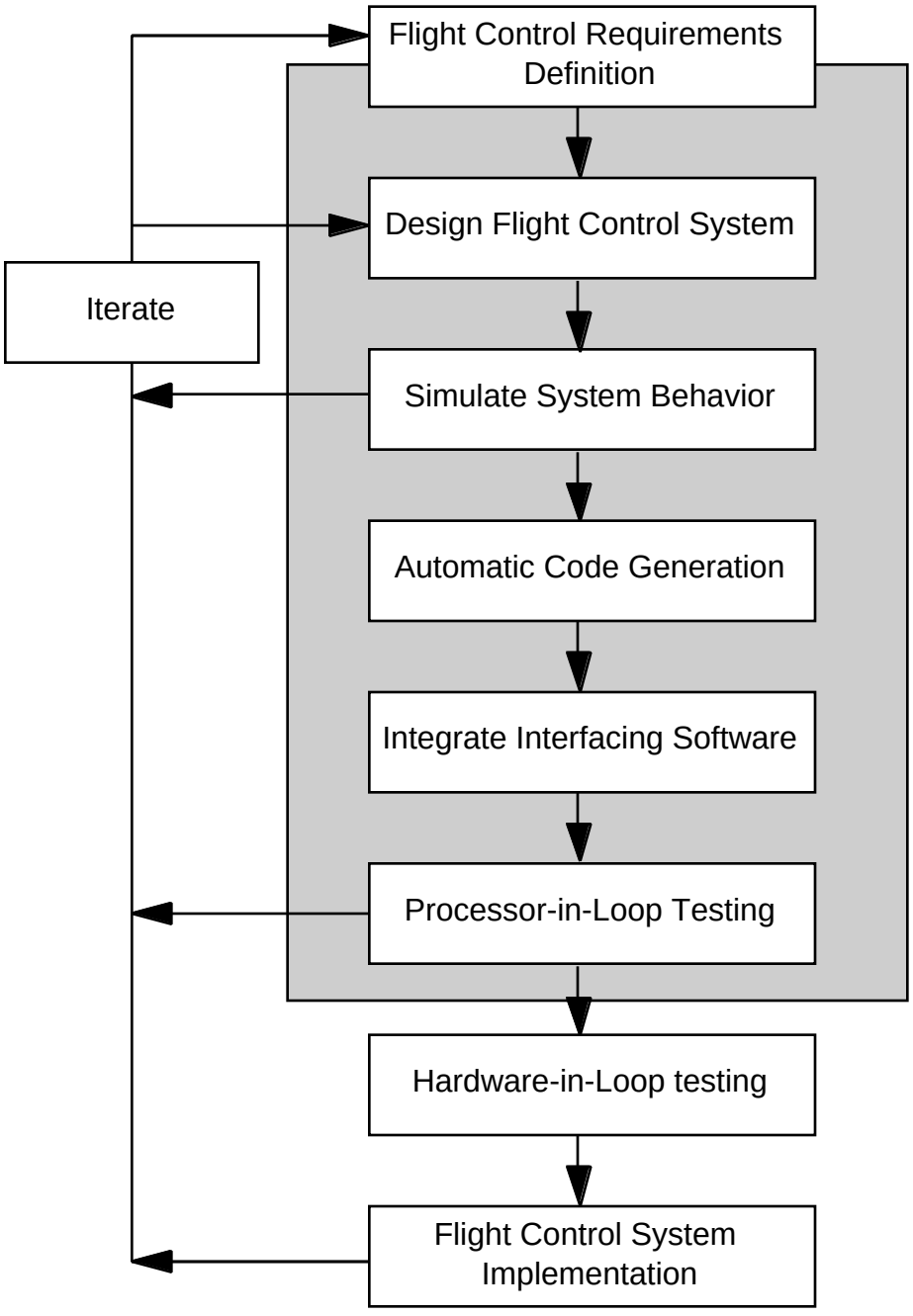
RAPIDS is a highly iterative process utilizing rapid prototyping techniques. Each rapid prototyping cycle develops a complete GN&C system from requirements definition through design and implementation on a target processor. Processor-in-loop (PIL) testing is performed once the initial working FSW prototype is developed.

This approach allows problems with software design, implementation, or hardware selection to be discovered early in the development cycle. This cycle of concurrent requirements and software development and PIL testing is repeated until FSW with the desired performance and quality is produced. The approach employed by RAPIDS is a spiral development approach where developers “build a little, test a little”. This process is illustrated in Figure 2.1.

Each RAPIDS design cycle involves phases which can be found in a traditional process—requirements, design, development, and integration and test—that are performed with varying degrees of concurrency. Cycle time decreases with each iteration while the quality of the FSW increases. Whereas a traditional approach may require years to complete a single design cycle, initial iterations of a RAPIDS process may require months and then only weeks as the design matures. Ultimately, the design cycle may be on the order of a few days or less. However, regardless of the length of the design cycle, configuration control and complete software validation testing are maintained.

A designer in a RAPIDS process is part of a small, integrated team and is involved for the whole design, development, and validation process. Using an integrated team of system designers means that the distinctions between systems engineering, GN&C engineering, and software engineering break down—a single team member may be asked to perform any of these functions over the course of a development program. Whereas in the past each functional discipline would own only a portion of the final design and only a certain phase of the program, the RAPIDS design team has ownership of the entire process and end product. Indicators are that end-to-end ownership of the product and process tends to be more efficient and promotes a more productive work environment for the designers (Maras et al., 1994).

Figure 2.1: RAPIDS Spiral Development Process



Note: Hardware-in-loop testing involves the actual flight hardware. Processor-in-loop testing is performed with commercially equivalent hardware.

2.1.3 RAPIDS Toolset

The RAPIDS toolset integrates requirements analysis through hardware and software testing in a workstation environment using an integrated set of commercial off-the-shelf software development and simulation tools. The graphical user interface toolset captures design details being implemented by GN&C experts and then automatically generates source code and documentation that can be targeted to the actual flight vehicle computer system. The environment is currently based on commercially available products including ISI's (Integrated Systems, Inc.) MATRIX$_X$/SystemBuild/AutoCode/AC-100^{TM} and Cadre Teamwork.

MATRIX$_X$/SystemBuild is a graphical software tool that enables users to develop data flow block diagrams of the desired system using elementary building blocks. These elementary blocks can be organized into "Superblocks" which become procedures or subtasks. This construction process yields highly modular software designs which can facilitate the development of generic software libraries and the reuse of software. After construction is completed, the software data flow diagrams can be interactively tested in a non-real-time environment. Time and frequency domain analyses can also be performed interactively.

The AutoCode tool can automatically translate the block diagram representations into FORTRAN, C, or Ada source code. The source code can then be integrated with other interfacing software—software necessary for the code to run on the target processor such as a real-time operating system device driver, compiled and run on the AC-100 real-time computer to verify real-time and PIL performance.

2.1.4 Benefits of the RAPIDS Process

The ultimate benefit of this approach is a cost reduction because of the smaller software development staff necessary to support initial requirements definition through test and integration. The reduction in software development staffing is possible since the application or GN&C designer does the majority of these activities within an integrated, graphical workstation environment. Other benefits of the methodology include the following:

- Software is not a schedule critical item, and the best design can be implemented at flight time because it is not limited by the typical six to twelve month lead time required by a traditional process to make software changes. The design that flies can be based on the best available data and algorithms from all previous testing.

- Challenging milestones, which are typical of fast paced programs, can be met while maintaining or enhancing the quality of the final software product.

- The rapid prototyping process allows major errors and design flaws to be discovered earlier in the program when they are cheaper and easier to correct.

- Requirements can be verified early in the development program.

- Metrics tracked during DC-X1 software development indicated that the RAPIDS process can result in productivity improvements greater than 25 percent.

The Navigation, Control & Aeronautics Division at NASA's Johnson Space Center employed a similar process to construct a simulation of the Soyuz Assured Crew Return Vehicle flight software and demonstrated substantial productivity gains when compared to COCOMO model estimates. The data are shown in Table 2.1.

Table 2.1: Soyuz Simulation Project Metrics

	Phase 1	Phase 2
Number of Superblocks	55	371
Number of SLOC	4102	25045[a]
COCOMO Estimated Total Staff-Hours	3400	11658
Estimated Total Staff-Hours	1830	7720
SLOC per Staff-Day	18	22[b]
Productivity Increase (Actual vs. COCOMO)	85%	50%

Source: Uhde-Lacovara et al. (1994)

a. This figure includes the lines of code produced in Phase 1.

b. Reuse of Phase 1 software is assumed. Thus, the number of SLOC from Phase 1 was not included in calculating this value.

2.2 RAPID DEVELOPMENT THE GRITTECH WAY

Recognizing that the rate of complexity growth in electronic hardware and software was rapidly outdistancing the ability of its engineers to keep pace, GritTech[5] began to experiment with rapid development processes and enabling technologies to determine if dramatic improvements in productivity could be made. The basic question addressed in formulating a rapid development process was how to change the traditional process to exploit more fully the potential of design automation tools.

At GritTech the traditional development process for a typical module involved 30 or more discrete steps which were performed in a sequential, isolated fashion. Experience with this process taught the designers that the traditional process would often lead to the propagation of flaws which would remain undiscovered until late in the development cycle where rework can be a most costly and time consuming process. For instance, a small misinterpretation of the customer's specifications on the part of the contractor early on in the development process could embed a flaw in the design that could escape detection until field tests are conducted at the end of the development chain. Corrective action in this case could very well be a lengthy and expensive process. Even when errors remain undetected for only a few steps in a sequential process, the result could be significant budget and schedule overruns.

GritTech's answer to the productivity problem was to search for development processes which tightly integrated the diverse tasks so that they could be performed in parallel with rapid feedback and feedforward of information among all tasks. They believed that productivity would increase dramatically if the person doing a task could see the impact of a contemplated change on the results of all other tasks within minutes, rather than months.

2.2.1 General Process Characteristics and Philosophy

The rapid development process is composed of several technical and procedural elements. The relative importance of an individual element depends on the type of development

[5] This is a pseudonym for a defense electronics firm's rapid development group whose practices were studied for this report. The pseudonym is being used, at the request of the firm, in the interests of preserving confidentiality.

project at hand. Reflecting a kind of "skunk works" approach to project management, the designers responsible for the undertaking are given a wide degree of latitude in tailoring the process to the needs of the current project.

In addition, designers involved in rapid development projects generally have more responsibility for the project than designers working on a more traditionally managed program. For instance, the actual system designers are often personally involved in meetings with the customer. Project managers are also more involved with the day-to-day work of the designers, often getting involved in actual design activities themselves. The combination of broader responsibility for designers and more involvement by project managers facilitates better assessment of the current status and rate of completion of project work.

The rapid development designers are all high caliber engineers. If some "ilities"[6] are not required to satisfy customer requirements and are tailored out of the official project process in order to meet tight scheduling constraints, the individual designers will still try to account for them informally as a part of exercising "good engineering practice". For instance, consider a technology demonstration project under such time pressure that tasks such as explicit design activities, which are meant to ensure a certain degree of expandability, are tailored out of the official project process. However, a project engineer may still include some spare pin locations on a board in order to easily accommodate the addition of more memory or processing power in case it becomes necessary to increase the functionality of the system in the future.

A major tenet of the rapid development operating philosophy is to utilize all available means to enhance productivity and to allow the designers to experiment with new technologies to accomplish this. This tenet manifests itself operationally in a number of ways. For instance, designers in the rapid development group at GritTech utilize the Internet to get advice from outside experts. Frequently, assistance and advice can be obtained for free from the many technically-oriented newsgroups on UseNet. Source code applicable to a project at hand can also be found via ftp (file transfer protocol) or gopher sites. This tenet has also manifested itself in the willingness of designers to experiment

---

[6] "ilities" is shorthand for a class of design considerations such as manufacturability, affordability, supportability, scalability, upgradability, producibility.

with and adopt new design tools. In several instances designers have even adopted new design tools and used them for the first time on actual projects that were already in process, believing that the tools would help them perform their tasks better and faster.[7]

Typical rapid development projects do not attempt to extend the state-of-the-art in electronic hardware and software technologies. Most involve exploiting state-of-the-shelf technologies in new ways to produce a desired capability or set of capabilities. Demanding schedules are part of the norm.

To reduce the risks involved with development projects, the group does not usually attempt projects unless it has had some prior experience with the technologies involved. In some cases the group has even conducted some rapid prototyping activities before submitting a bid to ensure that risks are understood. Sometimes even hardware components will be part of the rapid prototyping effort. To further reduce risks involved in a development project, rapid development engineers will use real data whenever it is available since simulated data can be imperfect.

2.2.2 Examples of Rapid Development

As previously mentioned, GritTech's rapid development process and operating philosophy allow engineers extensive latitude in tailoring the process to the particular needs of the project. Moreover, most of the projects performed by the rapid development group at GritTech do not involve product line systems. Hence, it is useful to briefly examine a couple of rapid development projects—one hardware/software system and one software application—to gain a better understanding of the methodology.

Hardware/Software System Development

A good example of the application of GritTech's rapid development philosophy to hardware/software system development involved the design and development of an acoustic processor. The objective was to take a power-hungry, computation intensive system and develop a portable, low-power system providing equivalent functionality. Since

[7] Another aspect of this constant search for new tools and methods is the group's resistance to standardization of tools and processes. According to rapid development engineers, the traditional usage of standardization—one in which tools and processes that are standardized remain the company's standard even though much better tools and processes may exist—overly constrains the ability of a designer to use the best available tools and methods.

the chosen low-power processor provided only a fraction of the original system's computational power, it was necessary to decrease the computational load of the algorithms without appreciably degrading the performance of the system. Adding to the challenge was a four month development schedule.

Since simulation models were not available for some components that had been chosen for use, the hardware design needed to be prototyped and verified before the actual printed circuit board was fabricated. While wire wrapping—the traditional prototyping technology—can be relatively inexpensive, it was not flexible enough for the project's demanding schedule. The wrapping of the initial design can take a week, and corrections to the design can be time consuming and are often not properly documented, which can cause significant problems during hardware debugging. Consequently, a new, flexible prototype technology was chosen—Field Programmable Interconnect (FPIC) devices and Field Programmable Circuit Boards (FPCBs).[8]

Utilizing the flexible prototyping technology, the prototype could be dynamically reconfigured based on changes to the schematic—a capability that proved its worth on many occasions. For instance, an easily-acquired SRAM (Static Random Access Memory) was used initially instead of the desired SRAM since the special, low-power SRAM selected for the design was unavailable at the start of hardware development. When the desired components were delivered, the design schematics were updated to account for the different packaging and pin layout, and the new netlist was downloaded to the FPCB within minutes. In the course of development, a bit-swap error was detected which could have forced an entire bus change and a one day delay if wire wrapping had been used. Instead, the FPCB-FPIC combination enabled a corrected design to be up and running within minutes.

The flexibility of the prototype technology allowed the hardware design to be completely debugged in one week without the aid of computer simulations. A conventional

[8] The FPIC is a commercially-available RAM-based passive routing device, containing over 900 usable input/output pins. The FPCB is a multilayer circuit board accommodating an array of pin sockets and a mounting area for one or more FPICs. A circuit is constructed by placing components on the board and configuring the FPICs. The FPIC also has a dedicated logic-analyzer diagnostic port. Extensive software tools for translating netlist information into component placement, FPIC internal routing, and logic-analyzer configuration information are also available from the vendor. FPIC and FPCB are trademarks of Aptix Corporation.

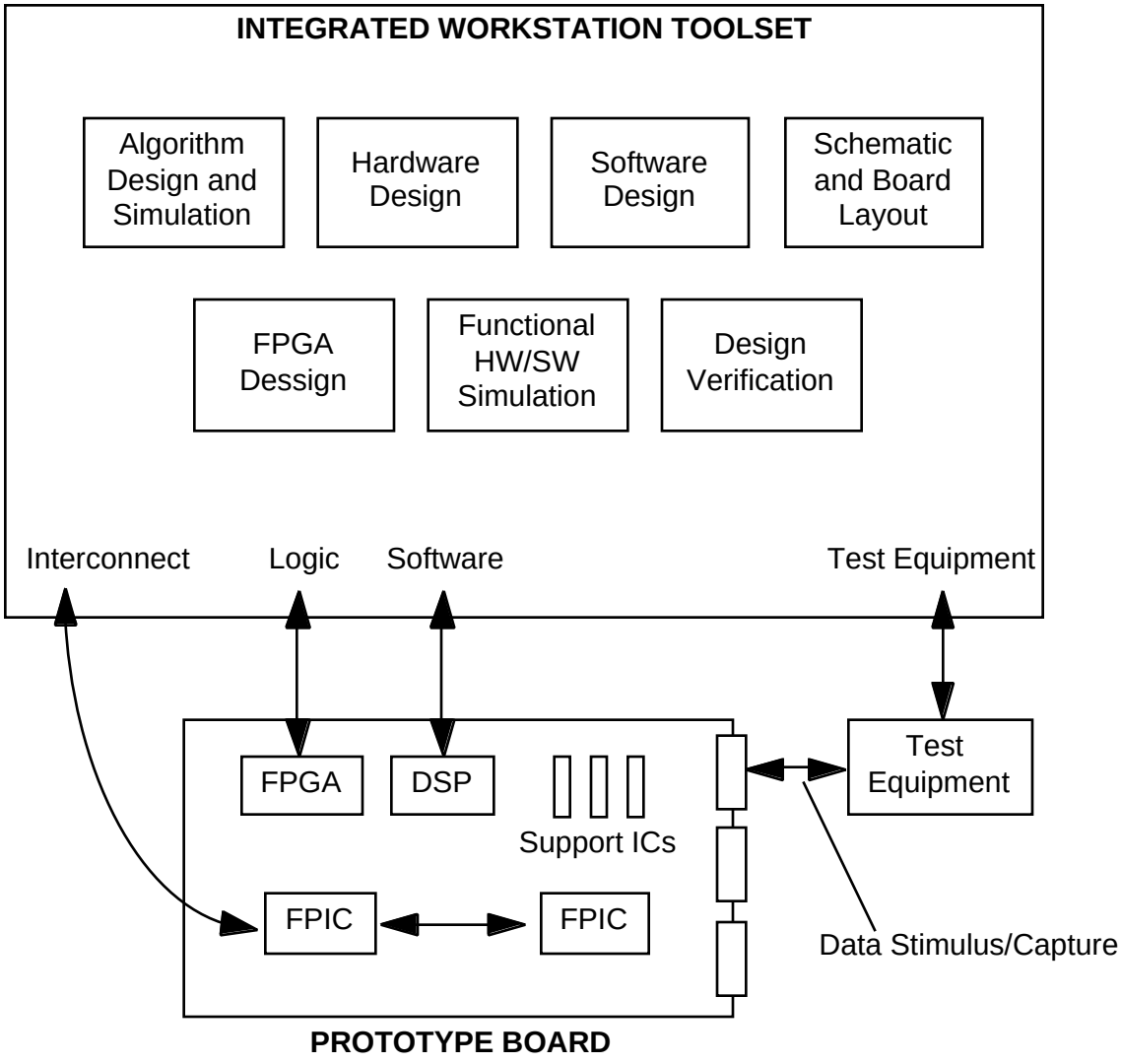
printed circuit board layout was performed concurrently with the hardware design verification effort, and schematic changes made as a result of the debugging effort were automatically included in the printed circuit board (PCB) layout. After only one week of testing, the PCB was shipped for fabrication using the identical netlist that had been validated on the prototype. When the bare PCB was delivered, the components were dropped in, and the system was thoroughly tested. After two days of testing, no defects were discovered, and the board was declared finished with no cuts or jumpers required. No problems have been reported in subsequent operational use.

While the PCB was being fabricated, the application software was being coded and tested using the integrated development environment, which included the FPCB-FPIC prototype, an in-system processor emulator, networked logic analysis instruments, a "reference design" from a previous project, and additional signal processing analysis tools running on a workstation. A depiction of this environment is shown in Figure 2.2. The combination of the ability to control the hardware and test equipment from the workstation by downloading and uploading code, data, and sequencing information and the ability to orchestrate the use of the various assets with operating system scripts provided designers with a powerful, integrated rapid development environment. Moreover, having a completed printed circuit board in a little more than one month enabled the designers to focus their efforts on developing more sophisticated algorithms during the remaining three months and achieve better field-test results.

In addition to the speed of development afforded by the integrated environment, application software development was further accelerated through the use of several routines that were obtained from bulletin boards on the Internet. These routines provided the needed throughput with minimal modification.

According to the designers, the success of the rapid development project could be attributed to the use of a combination of newer technologies which resulted in a major reduction in development time. Compared to estimates of traditional development effort based on a benchmark of 15 staff-months/board for typical module development productivity, the hardware/software effort required only 16 percent of the engineering staff-months. Thus, by combining an integrated development environment with

Figure 2.2: Integrated Development Environment



programmable prototype methods, the GritTech rapid development engineers achieved a factor of six improvement in productivity.

Incremental Software Development

Another example of rapid development in action was the use of the process in the development of a launch data visualization and advising system. This particular project arose from the customer's desire to exploit newly available data visualization capabilities for a launch vehicle program in 1991. Over 50,000 pressure, temperature, wind direction, and other sensors were in use during launch preparations. The data stream from these

sensors was being preprocessed and displayed to operating personnel in the form of instantaneous numeric values. The customer identified a need to display this information graphically, provide trend information at a glance, enable comparisons to past launches, and perform related functions.

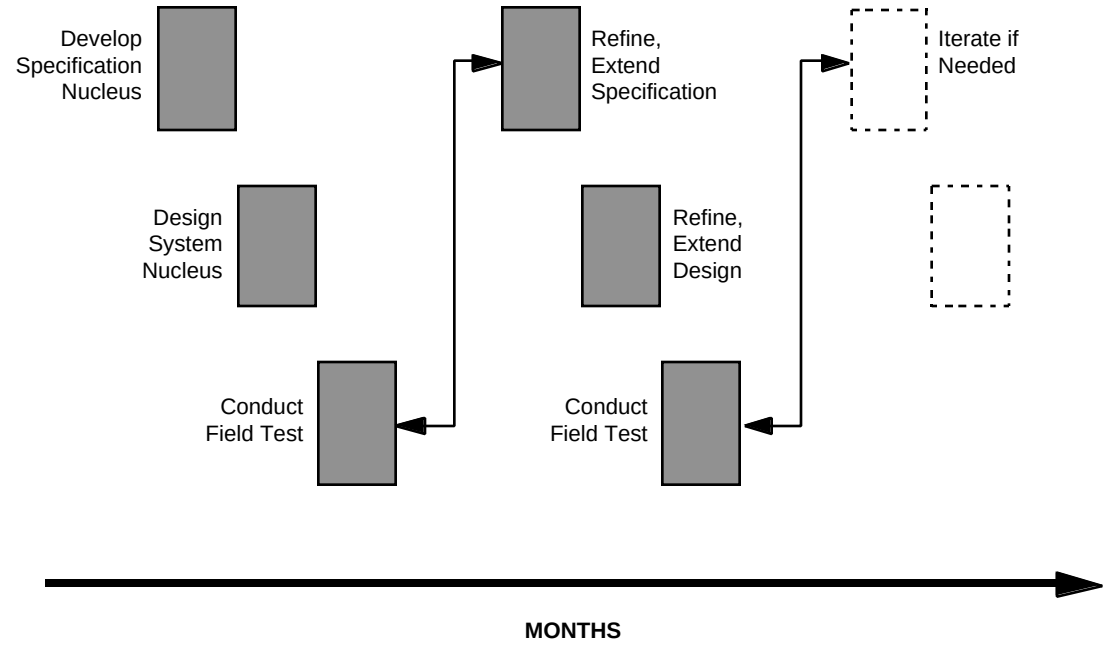
Rapid development engineers developed a core analysis and display system in about a month after project launch. This initial release was then installed for evaluation on computers at the launch facility, off-line from actual launch operations. Incremental release of the most recent version for customer evaluation occurred approximately every five weeks. Feedback included the suggestion to add capability to call up video views of the launch vehicle and the launch pad. The GritTech engineers discovered that the incremental development and release approach also helps to build a good working relationship with the customer in addition to speeding up the development process. Design cycles tend to get progressively shorter with each iteration, partly because the integration of software modules is performed many times and gets progressively easier.

The incremental development process is depicted in Figure 2.3. Incremental development utilizes a series of quick low cost field trials of progressively more complete systems. This approach differs considerably from the traditional approach which defers field testing until the end of a multi-year, full scale development program.

In the incremental development process, software design and development can begin as soon as some part of the system specification has been developed. Typically, the specification or portion of a specification is translated into a field-testable design in about a month. The customer is then supplied with a copy of the current software for operational or test range evaluations of the design with the GritTech designers providing support. This field testing is especially important for evaluating the design of user interfaces and displays. Based on the results of the joint evaluation, the specification is extended or revised, and the design is incrementally expanded and/or refined during the next cycle.

The rapid development group has found from past experiences that the optimum time period for providing customers with opportunities for hands-on evaluation is approximately once every four to ten weeks. The result is a design that rapidly evolves into a well-suited, highly functional system with minimal need to expend resources on

Figure 2.3: Incremental Development



performing rework resulting from erroneous assumptions and interpretations of requirements.

As was the case with the acoustic processor example, using a highly integrated set of software development tools facilitated a substantial reduction in the time and effort required for system development. One tool allowed software engineers to design the graphical user interfaces (GUIs) by manipulating basic GUI building blocks available from a palette. Once the elements were arranged to the satisfaction of the engineer, the actual source code was produced through automatic code generation. As a rule the generated code was not touched unless problems could be explicitly traced to it. Another tool which helped speed development enabled designers to execute the code and see how it worked without having to compile it first. Software libraries also facilitated the reuse of previously developed and verified software modules.

Since the launch data visualization system uses virtually all commercial off-the-shelf workstation and video hardware, development productivity was only measured with respect to the software development benchmark of 10 standard lines of code per staff-day. By using a highly integrated suite of software development tools in conjunction with an incremental development approach, the rapid development team was able to develop this

system and demonstrate an 8:1 improvement in productivity in spite of having to absorb a significant revision to the performance requirement.

2.2.3 Rapid Development Productivity Performance

To date, the rapid development process has been used on 20 different small to medium sized projects at GritTech, demonstrating significant productivity improvements over a traditional process in each case. The improvements in productivity afforded by the rapid development methodologies and tools for these projects are summarized in Table 2.2 and Table 2.3 for software and hardware development, respectively. The rapid development projects have consistently exhibited two to four times the productivity that would be expected of a traditional process.

Table 2.2: Software Rapid Development Results

Environment	Productivity Improvement (versus benchmark)
Tailored DoD-STD-2167A	3 to 4:1
Other	3 to 8:1

Table 2.3: Hardware Rapid Development Results

Module Type	Type	Productivity Improvement (versus benchmark)
6 inches x 9 inches (54 square inches)	Microprocessor-Based	2 to 6:1
8 inches x 16 inches (128 square inches)	Logic-Based	2 to 4:1

2.3 HARDWARE/SOFTWARE CODESIGN

Methodologies to support the codesign of hardware/software systems were developed in response to problems with the traditional process for developing these systems. The traditional process partitioned the problem into hardware and software elements early in the development cycle, and then proceeded to develop the two designs in parallel with very little or no interaction until the end of the process when they were integrated for system

testing. Predictably, the classic approach exhibits many symptoms of a fragmented process.

Typically, the integration of the hardware and software near the end of the development cycle is laden with unforeseen problems—many of which can be attributed to the lack of interaction between the hardware and software design groups. Any design changes at this point in the process are likely to significantly impact the system's cost and development schedule. In many instances, even when integration itself does not reveal any problems, the overall system performance can be disappointing. Frequently, the blame is laid at the feet of the programmers. However, in many cases, the real fault may lie in the design of a development process which imposes an artificially crisp distinction between hardware and software design.

2.3.1 Generic Hardware/Software Codesign Process

An alternative approach is to recognize the high degree of coupling that exists between hardware and software for most complex electronic system design problems and employ a more flexible design process, where hardware and software development proceed in parallel with feedback and interaction between the two as the overall system design matures. The final hardware/software partitioning decision can be made after evaluating alternate design architectures with respect to such factors as performance, programmability, reliability, and manufacturability. This type of approach may be termed "hardware/software codesign".

According to Kalavade and Lee (1992), hardware/software codesign strategies can be applied to different levels of design problems including:

- *Processor Design.* An optimized application-specific processor can be developed by tailoring both the instruction set and the program for the application. This type of codesign problem is very difficult.

- *System-Level Design*. Hardware/software codesign can also be performed at the system level, where an algorithm is partitioned between custom hardware and software running on programmable components. The hardware would typically include discrete components, application-specific integrated circuits (ASICs),

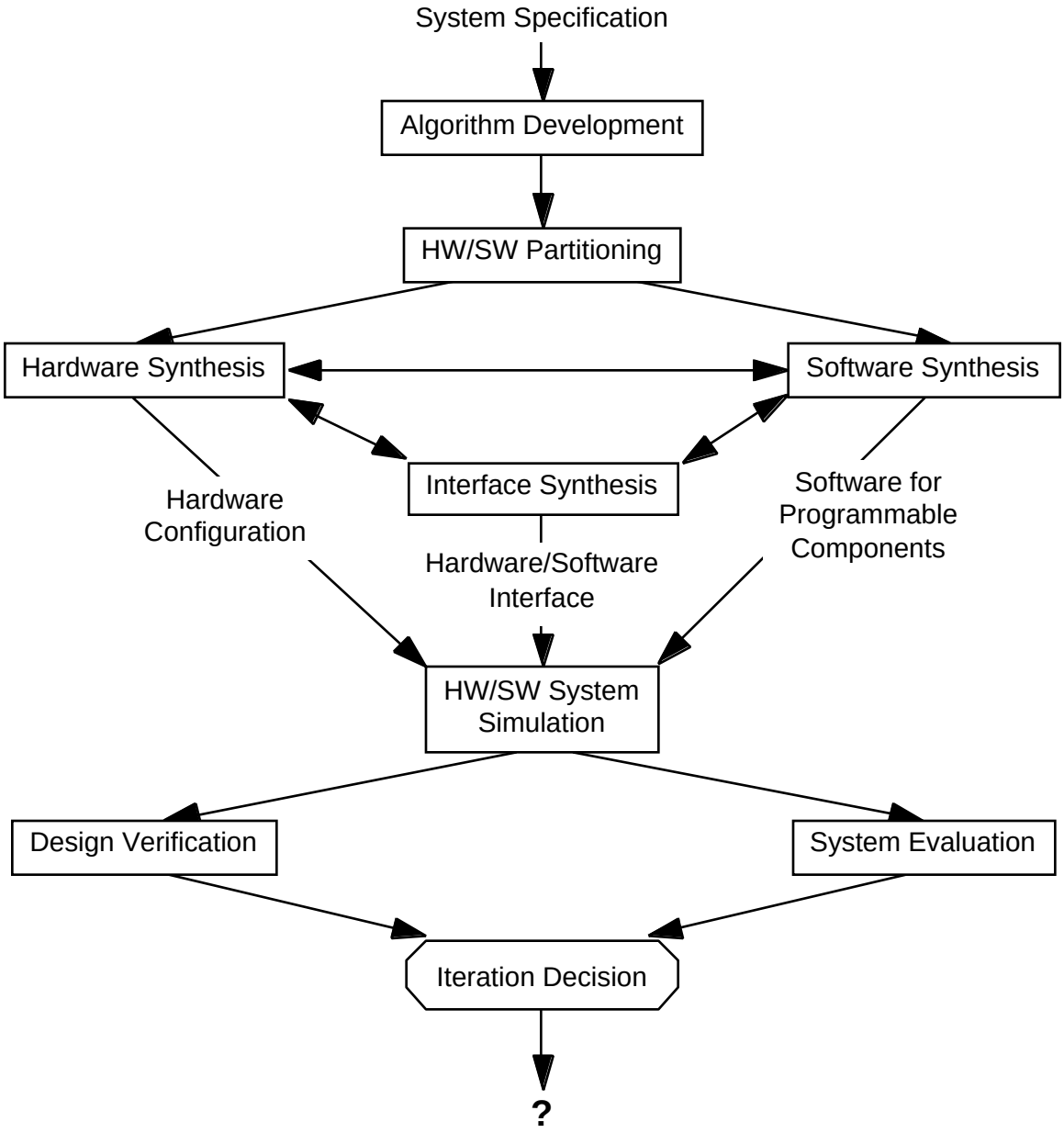
DSP cores, microprocessors, microcontrollers, or semi-custom logic developed using FPGAs or logic synthesis tools. Since there are many possible ways to partition a given design between hardware and software components, evaluating design configurations with system-level simulation of hardware and software allows the design space to be more thoroughly explored and is an integral part of codesign.

- *Application-Specific Multiprocessor System Design.* The codesign of an application-specific multiprocessor system is challenging since it involves choosing a suitable number of processors, an interprocessor communication (IPC) strategy, and the design of the application software. Since software synthesis requires partitioning and scheduling the code among the processors, scheduling techniques must be capable of adapting to changing hardware configurations. Thus, developing an application-specific multiprocessor system is an iterative process, involving tradeoffs associated with selecting an optimal hardware configuration and software partitioning.

A generic hardware/software codesign process is shown in Figure 2.4. The objective of a codesign methodology is to produce a hardware/software system design that meets a given set of specifications while satisfying a set of design constraints. Given a system specification, a designer can utilize high-level functional simulations to develop a suitable algorithm without making any assumptions concerning specific implementation details. The next step is to partition the algorithm into hardware and software while satisfying requirements such as speed, complexity, and flexibility. Operations that are computationally intensive with fixed operations are usually allocated to hardware. Algorithm components that may vary for different situations, are less computationally intensive, may require field programmability, or are not likely to change with time can be allocated to software (Kalavade and Lee, 1993).

Once the initial partitioning has been performed, the process of synthesizing the hardware, software, and interface designs can begin. These three activities are tightly coupled. Changes in one synthesis area significantly affect the others. Hardware synthesis activities include selecting the programmable processor, which directly impacts the

Figure 2.4: Generic Hardware/Software Codesign Process



software synthesis activity of selecting a code generator, and determining the appropriate number of processors and their connectivity, which, in turn, influences the code partitioning decision and hardware/software interface synthesis. Choices involved in custom hardware synthesis can range from generating custom data paths to generating

masks for FPGAs. As a part of custom data path design, the register word lengths must be selected (Kalavade and Lee, 1993).

Depending on the chosen hardware configuration, software synthesis can involve partitioning and scheduling the code across multiple processors and synthesizing the code for interprocessor communication—decisions which depend heavily upon the selected architecture. Partitioning among different processors may be performed with the intent of optimizing cost functions such as communication cost, memory bandwidth, and local and global memory sizes. In addition, if the hardware configuration includes use of fixed-point processors, some algorithmic modifications might be required to minimize finite precision effects, such as limit cycles and quantization errors (Kalavade and Lee, 1993).

Interface synthesis involves adding latches, FIFO (first in, first out) registers, or address decoders in hardware and adding code to handle input/output operations and semaphore synchronization in software. Iterating to explore different design options is the common method for solving this cyclic problem (Kalavade and Lee, 1993).

After the hardware, software, and interface synthesis tasks have been accomplished, the hardware/software system design can be simulated within a heterogeneous simulation environment. Since the simulated hardware must run the generated software, the simulation environment should allow for the interaction of a number of different simulators in the event that various specification languages are used.

Simulation results can then be used to verify that the design works as intended and meets the given system specifications. If the specifications are not satisfied, another iteration will be needed. Whether it is necessary to perform another iteration of the entire codesign process or just portions of the process will depend on the nature of the shortcoming. The simulation results and the hardware and software configurations chosen for the specific system design can also be used to evaluate the system in terms of performance and estimates of other factors including power requirements, die area, component and bus utilization, and manufacturing costs. After using these estimates to evaluate the design, the designer may choose to repartition the system and experiment with different designs (Kalavade and Lee, 1993).

Currently, there are several programs that are developing design environments which support hardware/software codesign or are attempting to incorporate those ideas into

a larger system engineering methodology while developing the enabling tools. The next two sections provide a sample of these efforts.

2.3.2 The Ptolemy Project

Developed at the University of California at Berkeley, Ptolemy is an environment for prototyping and simulating heterogeneous systems.[9] Heterogeneous systems are systems involving subsystems having different models of computation and, hence, fundamentally different approaches to design and simulation. According to Dr. Mark Richards of ARPA, Ptolemy's framework allows a designer to mix and match multiple models of computation more effectively than most design systems. Ptolemy facilitates the interaction of diverse models of computation through the use of object-oriented principles of polymorphism and information hiding.

Since the start of the Ptolemy project in 1990, there have been numerous advances in design, simulation, and code generation. Many of these advances have been incorporated in Ptolemy in the realms of dataflow modeling of algorithms, synthesis of embedded software from such dataflow models, animation and visualization, multidimensional signal processing, hardware/software partitioning, VHDL (VHSIC Hardware Description Language) code generation, and managing complexity through the use of higher-order functions.[10]

Ptolemy employs object-oriented software principles in attempting to achieve the following goals (Buck et al., 1994):

- *Agility*. The Ptolemy environment should support distinct computational models to enable each subsystem to be simulated and prototyped in a manner that is appropriate and natural to that subsystem.

[9] The Ptolemy software is available for the Sun 4 (sparc), DecStation (MIPS), and HP-PA architectures. System installation requires 90MB of disk space and at least 8MB of physical memory. A scaled-down demonstration version, called Ptiny Ptolemy, is also available for the same architectures but only requires 12MB of disk space. A copy of Ptolemy can be obtained on tape by calling (510) 643-6687 or via anonymous ftp from ptolemy.eecs.berkeley.edu. Other on-line information resources can be found in the newsgroup comp.soft-sys.ptolemy and on the World Wide Web at http://ptolemy.eecs.berkeley.edu.

[10] The project joined ARPA's RASSP program, the subject of the next section, in 1993 as a technology base developer.

- *Heterogeneity*. Ptolemy should enable distinct computational models to coexist seamlessly in order to investigate interactions among subsystems.

- *Extensibility.* Ptolemy should support seamless integration of new computational models and allow them to interoperate with existing models with no modifications to the Ptolemy environment or existing models.

- *Friendliness.* Ptolemy should employ a modern graphical interface with a hierarchical block diagram style of representation.

Ptolemy has been used for a wide range of applications including signal processing, telecommunications, parallel processing, wireless communications, network design, radio astronomy, real-time systems, and hardware/software codesign.

In the hardware/software codesign application area, Ptolemy is a very powerful tool since all parts of a hardware/software system can be modeled using the various domains included in the environment. Modeling both hardware and software within a single framework allows a designer to explore tradeoffs between hardware and software implementations of different functions (Buck et al., 1994). Another advantage of being able to develop hardware and software simultaneously in a design and simulation environment is that software development does not have to wait for a hardware prototype to be ready in order to start. In fact, if the manufactured hardware configuration is the same as the simulated hardware/software system, the actual production software code will have already been developed and verified.[11]

2.3.3 Rapid Prototyping of Application Specific Signal Processors[12]

Initiated in 1993, the Rapid Prototyping of Application Specific Signal Processors (RASSP) program is a four-year ARPA/Tri-Service initiative aimed at creating a new process for the development of military signal processors. The program's objective is to

[11] A good example of how Ptolemy can be used in hardware/software codesign to explore the design space can be found in Kalavade (1991).

[12] Although this program is really just getting started, RASSP's concepts merit discussion since they provide a glimpse of design methodologies and capabilities that could be available in the near future. In fact, according to Dr. Mark Richards, RASSP program manager, substantial progress has already been made during the program's first year. This description is based on many interviews (live, phone, and email) with Dr. Richards and others at ARPA as well as the papers referenced herein.

dramatically improve the process for the development of complex digital systems—particularly embedded digital signal processors[13]—in the areas of specification, design, documentation, manufacturability, and supportability.

Major Goals

The major goals of the RASSP program are:

- 4x reduction in concept-to-fielding cycle time
- Commensurate improvements in quality, life cycle cost, and supportability
- State-of-the-art at the time of fielding
- Systematic design capability at all levels from concept to manufacture
- Commercialization and promulgation of the RASSP process

RASSP Approach

The RASSP approach has three key thrust areas—design methodology, processor architecture, and design automation. The first key thrust is an incremental refinement of the design methodology. As currently envisioned, the design methodology will be based on concurrent engineering practices, using a top-down, VHDL-based design approach. The methodology will also seek to involve users early and often throughout the design process.

The second thrust is in the area of processor architecture. RASSP will foster the use of modular hardware and software architectures through the separation of communication, computation, and control functions and scalable interconnects.

The final thrust is the development of a seamlessly integrated and comprehensive set of CAD tools. Tools to support all the "ilities" are to be included as well as manufacturing and system engineering.[14] The design environment would also facilitate true hardware/software codesign and hardware and software synthesis. Hardware and software

---

[13] While signal processors were chosen as a focal point for the RASSP program, it is hoped that the processes and tools developed during the program will be broadly applicable to the domain of complex digital systems.

[14] "ilities" is shorthand for a class of design considerations such as manufacturability, affordability, supportability, scalability, upgradability, producibility.

reuse libraries and an enterprise framework would also be incorporated into the EDA (Electronic Design Automation) infrastructure.

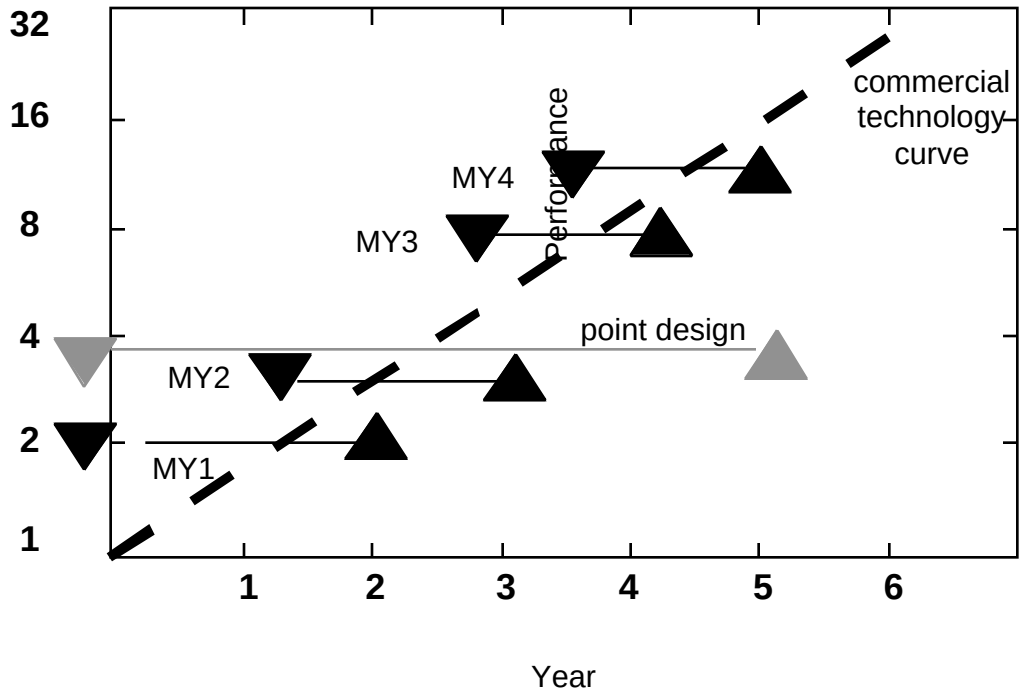
RASSP Design Methodology

Two major components of the design methodology being explored by the RASSP developers are top-down concurrent design and the model year concept of design. The RASSP design methodology is derived from a top-down approach to system engineering, starting with a formal specification which is successively refined to finer and finer levels of detail until the design is finished. Concurrent engineering concepts are applied to modernize the traditional top-down approach and are expected to contribute significantly to the goal of reducing design cycle time by a factor of four. Through the use of EDA tools, RASSP hopes to be able to provide the designer with estimates of factors such as size, weight, power, cost, and reliability early in the design cycle while the designer is still experimenting with different algorithms and system architectures.

The RASSP design methodology is also experimenting with a model year approach to design. Traditionally, DoD has emphasized the development of the best possible technology at the time of initial concept development. The development of military signal processors is no exception. The result of this point design approach is usually the development of highly optimized custom hardware, interfaces, and software in an attempt to maximize performance. Moreover, insisting on the use of custom design requirements frequently results in very long development cycles. Not only does this introduce a significant delay in the delivery of prototypes to potential users for operational evaluation, but it can also lead to the procurement of systems that are already obsolete by the time they are fielded. Typically, upgrading point designs is also a difficult and expensive proposition because of the highly optimized design of the original system.

In contrast to this traditional approach, the model year design methodology—derived from the practices of top commercial electronics companies—primarily utilizes existing hardware and software technology to rapidly develop a baseline system based on a subset or relaxed set of specifications. The resulting prototype can be delivered to the user for test and evaluation earlier than with the traditional approach, which also allows the designers to get user feedback faster. Subsequently, the design can be upgraded to correct

Figure 2.5: Model Year Design vs. Point Design



any functional problems and insert more recent technology into the system. Over the length of time required to field a point design, a hardware/software system designed according to the model year philosophy may evolve through several design cycles (Richards, 1994). Thus, the model year approach to design can place a capability in the hands of the user earlier and enable the fielded system to keep pace with the rate of technological advance. A conceptual depiction comparing the point design and model year design approaches is shown in Figure 2.5.

As illustrated in Figure 2.5, the model year methodology assumes the performance of commercially available technology improves rapidly. As long as commercial technology continues to improve rapidly, the successive refinement of a system design through the use of several short design cycles will produce better performance than the point design approach even if the original model year design sacrifices some desired performance. Moreover, the improved level of performance can be achieved with a markedly decreased dependence on costly, hard-to-maintain custom hardware and software (Richards, 1994).

Use of a model year approach also creates a number of benefits for the user. First, a higher-performance signal processor can be acquired for a lower design cost. Second, since the processor is based on standard supportable technology, life cycle support costs are lower. Evolving model year prototypes supply the user with prototype hardware and software early and often, providing a means for discovering flaws in the system specifications while they are still correctable. The result is a product that is well suited to the needs of the user (Richards, 1994).

RASSP Architectural Concepts

The RASSP architectural concepts do not prescribe a specific processor design. Rather, they form a flexible framework for DSP design and provide architectural guidelines to be observed in order to realize the full potential of the model year design paradigm. The architectural concepts include scalability, modularity, flexible interfaces, heterogeneity, and life cycle support (Richards, 1994).

- *Scalability*. Although the program is focusing on the embedded DSP domain, the performance range that must be addressed is still sizable. To meet the objectives of the RASSP program, architectural ideas should be scalable to satisfy performance requirements ranging from a few megaflops to tens or possibly hundreds of gigaflops.

- *Modularity*. An efficient model year development process cannot be achieved unless each succeeding design cycle is able to build upon the work of previous cycles. Thus, processor architectures must facilitate the design and reuse of hardware and software in a modular fashion to allow portions of the processor to be upgraded without a wholesale redesign of the system.

- *Flexible Interfaces*. The processor subsystems should employ interfaces based on scalable, open hardware designs and software communication protocols. Since the interfaces to the actual sensor and to displays or data processors will generally be beyond the control of the RASSP designer, flexibility in interface capabilities will be a must. The combination of modularized hardware with standard interfaces between modules will localize the impact of design changes to the portions of the system being redesigned or upgraded.

- *Heterogeneity*. A key point is that RASSP is not biased toward any one particular implementation technology, such as ASICs or programmable devices. Instead, it is assumed that the RASSP design system must be able to handle combinations of custom, hardware programmable, and software programmable implementation techniques. In terms of the hardware mix, a RASSP processor could include custom ASICs, FPGAs, and a fully programmable embedded processor composed of commercial off-the-shelf DSPs, RISC processors, and high performance computing modules. This mix of computing elements illustrates the need for architectures (and a design system) capable of managing heterogeneity.

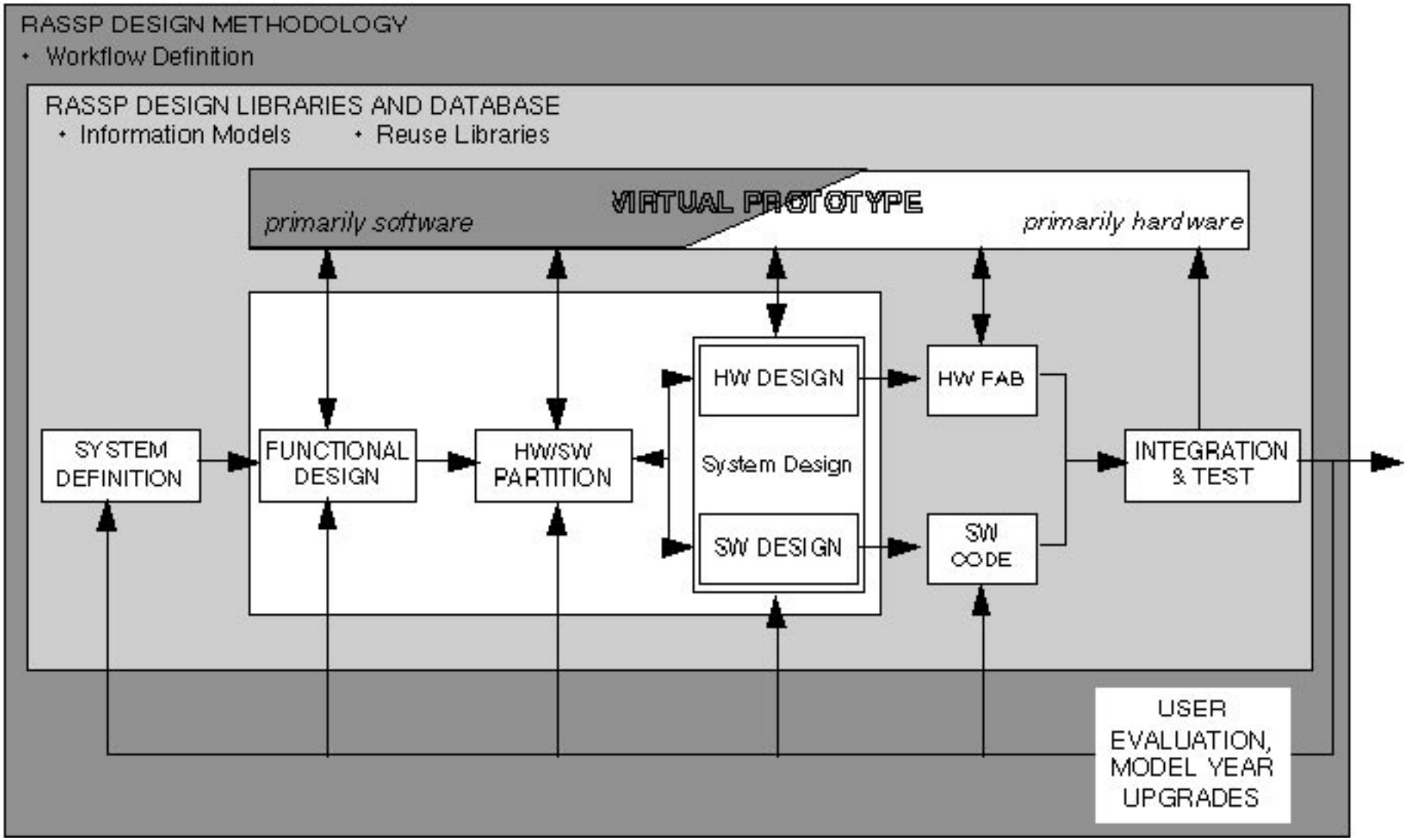
- *Life Cycle Support*. This architectural requirement addresses concerns related to upgradability and testability. The use of modular hardware and software and flexible interfaces within a model year design framework will ensure that the system will be upgradable. RASSP processors will be designed for extensive hierarchical testability from the start as part of the concurrent engineering thrust to improve product quality and reduce the cost of field maintenance and support.

RASSP Development Environment

As currently envisioned, the RASSP development environment will support the designer from the earliest phase of requirements capture to the most detailed board and ASIC design, enabling full requirements traceability, virtual prototyping, and a smooth transition into manufacturing. A streamlined depiction of the RASSP design flow is shown in Figure 2.6. The development process starts with requirements capture during the system definition phase. RASSP is currently investigating the feasibility of using VHDL-based simulatable specifications as a portion of the processor requirements specification. A VHDL specification would implicitly represent both hardware and software functionality and could then be expanded, refined, and partitioned into explicit hardware and software modules.[15] Ultimately, the VHDL specification could be used as an input to synthesis tools

[15] Technology base contractors have already released a version of a VHDL-based specification language incorporating area, speed, and power constraints. In addition, work has already begun on developing VHDL libraries of common microprocessor models.

Figure 2.6: RASSP Design Flow



or reuse libraries. In addition, a simulatable specification could be used to specify a test bench at the highest level of system definition. More detailed tests could then be derived from this "master test" as the design matures while maintaining traceability to the original system-level test (Richards, 1994).

After the system requirements are specified, a functional design of the system comprising algorithms and control is developed to satisfy the requirements without allocating specific functions to hardware or software. The next step is to partition the functional design into hardware and software implementations. At this point major architectural tradeoffs are investigated to determine how the functionality should be divided between hardware and software and how the allocated functionality should be implemented, such as which parts of the algorithm need to be implemented in FPGAs rather than in code on an embedded processor or which parts of the software must be written in assembly language instead of a higher-order language. Other tradeoffs could involve choosing between a single or a multiple processor configuration or choosing a particular microprocessor or DSP for use in the system. These architectural choices will have major downstream implications for performance, cost, and supportability. Once the partitioning of the system design into hardware and software has been completed, the hardware and software designs are then refined and evaluated. The hardware/software codesign process—functional specification, partitioning, and hardware and software design and evaluation—is iterated as required in order to obtain a solution which meets performance requirements while satisfying constraints such as cost, form factor, and power. To this end, tools capable of providing early estimates of performance, cost, and physical characteristics need to be available to the designer to enable the development of a robust processor design (Richards, 1994).

The hardware/software design process is augmented through the use of design databases and virtual prototyping. In order for the model year design methodology to be both efficient and effective, synthesis and reuse of both hardware and software modules must be promoted and facilitated by the design environment. This requirement implies the need for a comprehensive database system capable of displaying multiple views of the design data throughout the development process. To further enhance the efficiency and effectiveness of the design process, virtual prototyping is used to explore design options

and ensure specifications compliance and traceability. The virtual prototype is comprised entirely of software during the early stages of the process, but increases in hardware content as the design matures. Advanced hardware and software co-simulation technology, such as the ability to mix different models of computation and the ability to allow simulation tools, emulators, and hardware interact through simulation backplanes, is required for the potential of virtual prototyping to be realized (Richards, 1994).

A full-fledged RASSP development environment will also include many high-level systems engineering tools not shown in Figure 2.6. Examples of such tools include workflow management tools, cost models, advisors to assist the designer in making early architectural decisions, “ilities” analysis tools to aid in evaluating designs for such concerns as reliability and manufacturability, and documentation and report generation tools (Richards, 1994).

Although current computer-aided design and computer-aided software engineering tools for hardware and software development are relatively mature, integrating these tools across design levels and vendors is still a non-trivial task. Further, while some tools exist for higher level tasks such as requirements capture, functional specification, and algorithm development, they are not as mature as the lower level tools or as well integrated with one another as the lower level tools. In addition, efforts to integrate these higher level tools with the lower level tools are in their infancy. Consequently, while the integration of the lower level tools will certainly be addressed, the development, refinement, and integration of the high level system engineering tools will likely engage a larger portion of the program’s efforts (Richards, 1994).

2.4 CLEANROOM SOFTWARE ENGINEERING

Early software development was based entirely on craft practices, characterized by a reliance on trial-and-error methods and the talents of individual programmers. This sort of development, which is still in use in many organizations, designs software in an unstructured, bottom-up manner. The conventional wisdom is that no program can be developed defect free. Rather, debugging is the standard method of getting software to work properly. The ensuing rapid growth in demand for additional functionality has led to

tremendous growth in the complexity of software designs, complicating the already difficult task of writing correct programs.

A significant advance in software development capabilities came with the advent of structured programming techniques in the 1970s. This improvement was precipitated by Dijkstra (1969) who advocated eliminating the use of GOTO statements and restricting control logic to just three forms: sequence (begin-end), alternation (if-then-else), and iteration (while-do). The arbitrary branching allowed by the GOTO statement provided programmers with great freedom in designing control structures and was considered to be the mainstay of programming ingenuity and creativity (Mills, 1986). However, this freedom was a double-edged sword. Undisciplined use of GOTO statements frequently resulted in the development of "spaghetti code"—programs with extremely complicated control structures. Up to this point, writing spaghetti code seemed to be necessary to satisfy the demands of the customer, and the three control structures appeared inadequate compared to the power of GOTO statements. Rank and file programmers were surprised to discover that the control logic of any flowchartable program—even spaghetti code—could be replicated by utilizing combinations of the three primitives. Furthermore, in contrast to spaghetti code, the structured programming approach defined a natural hierarchy among its instructions (Mills, 1986). Adopting this approach allowed developers to write structured programs in a top-down fashion. Many organizations also implemented code reviews to augment debugging. Implementation of structured programming produced significant quality and productivity improvements over the traditional trial-and-error methods, as shown in Table 2.4.

However, even with the aid of structured programming practices, software development remains largely dependent on craft-based practices. Conventional methods can be categorized as follows (SET, 1993):

- *Process-oriented*. These methods were strongly influenced by the sequential flow of computation supported by traditional programming languages, such as COBOL and FORTRAN. Identifying the principal processes of the system and the data flows among these processes is the focus of these methodologies. Yourdon's Structured Analysis and Structured Design is an example of a process-oriented system development method.

- *Data-oriented*. These methods are based on the importance of data files and databases in large business and industrial applications. In data-oriented methodologies, system processes are designed to support the data processing requirements of the application. Information Engineering is an example of a data-oriented method.

- *Object-oriented*. These system development methods focus primarily on objects and classes. A system is developed by designing the interaction of objects—identifiable entities encapsulating states and behavior—to generate a desired outcome. Coad-Yourdon object-oriented analysis and design is an example of this type of methodology.

Unfortunately, the use of these methods has not significantly elevated software development above its craft origins. Similarly, the introduction of advanced SEEs (software engineering environments), new life cycle models, and maturity models have yielded some improvements in quality and productivity, but not the quantum leaps that were hoped for. While these innovations are important, they do not address the root cause of the problem—the lack of a science base for software development (SET, 1993).

Cleanroom engineering addresses the quality and productivity problems by applying rigorous practices to achieve intellectual control over the project. It also establishes an "errors are not acceptable" attitude and makes quality a team responsibility. Furthermore, the software's reliability is certified through the application of statistical quality control methods. When compared to traditional and structured programming methods, the number of defects encountered during development and operational use is dramatically lower for software developed with Cleanroom engineering practices. Moreover, Cleanroom practices yield major improvements in productivity. Data comparing the defect densities and productivities of traditional, structured programming, and Cleanroom practices are shown in Table 2.4.

2.4.1 Fundamental Principles of Cleanroom Engineering

Cleanroom engineering, developed at IBM's Federal Systems Division in the 1980s, is named after the cleanrooms used to fabricate VLSI (Very Large Scale Integrated) circuits.

Table 2.4: A Comparison of Software Development Practices

Development Practices	Development Defects/KLOC	Operational Defects/KLOC	Productivity (LOC/Staff-Month)
Traditional	50-60	15-18	unknown
Structured Programming	20-40	2-4	75-475
Cleanroom	<5	<<1	>750

Source: SET (1993)

Like its namesake, Cleanroom engineering has no tolerance for defects. The ultimate objective of Cleanroom software engineering is to develop error-free software that functions perfectly from the first time it is tested to the end of its operational life.

The following sections describe four fundamental principles behind Cleanroom engineering: defect prevention, intellectual control, separation of development and testing, and certification of the software's reliability.

Defect Prevention

Cleanroom engineering espouses a "get it right the first time" attitude. Defects are not the result of bugs in the source code. Defects are the result of faults in one or more aspects of the development process. In fact, if the defect density of the software under development exceeds five defects per thousand lines of code, the offending software is discarded. The defects are examined to determine how the process failed and how the process can be improved to prevent the failure from recurring. The improved process is used to develop replacement software.[16]

Intellectual Control

Intellectual control is the ability to clearly understand and describe the present problem at the desired level of abstraction (SET, 1993). Schedule overruns, cost escalation, high defect densities, and the labor-intensive, craft nature of current software development practices are all symptoms of a process that is not under intellectual control. To achieve intellectual control, engineers must be equipped with theoretically sound intellectual tools

[16] Dr. Harlan Mills, one of the originators of Cleanroom engineering, has suggested that perhaps the most important tool for Cleanroom is the wastebasket (Mills and Poore, 1988).

and processes that give them a high probability of producing a correct solution. Enabling engineers to achieve and maintain intellectual control over projects is at the heart of the Cleanroom philosophy.

Separation of Development and Testing

Cleanroom engineering principles dictate that development and testing functions must be separated. Designers are not allowed to execute or test their own code. Only testers can compile and executed the software being developed. This may seem counterintuitive at first, but the reasons behind the separation are sound. First, since the developer cannot rely on debugging as a development crutch, he or she will focus more attention on writing correct software, rather than on finding and fixing errors. Second, debugging frequently imbeds deeper errors that are difficult to detect. Adams (1984) studied every failure report for nine of IBM's most widely used software products over several years and tracked each to its origin. In most cases the cause of the failure had been introduced by a fix for another failure.[17] The defects introduced by the fixes may not be found until the integration testing phase or during operational use by the customer. Debugging tends to produce software that is locally correct but globally incorrect.

Reliability Certification

Software reliability certification in Cleanroom engineering provides scientifically valid data that can be used to write warranty statements for software product quality. Reliability certification also provides feedback to development and management regarding the effectiveness of the software development process. Software is not released unless it meets or exceeds the mandated level of reliability.

2.4.2 Cleanroom Engineering Practices

The concepts expressed in the principles of Cleanroom engineering are reflected in its practices. The following sections describe some of the key practices behind Cleanroom engineering.

---

[17] The problem of bad fixes is well documented. In addition to the Adams study, Endres (1975), Fagan (1976), Jones (1978), Myers (1976), Shooman (1983), and Thayer (1978) all address the topic.

Structured Data

Increasing demands for software capability have resulted in the explosive growth of a data flow jungle that is just as tangled as the control flow jungle that existed before the advent of structured programming. Since arrays and pointers represent arbitrary access to data just as GOTO statements represent arbitrary access to instructions, Cleanroom practices recommends that randomly accessed arrays and pointers should not be used (Mills, 1986). These data structures should be replaced with such structures as queues, stacks, and sets. These structures are considered safer since their access methods are more disciplined (Head, 1994). In addition, while it may take more thinking to design programs without arrays, the resulting designs are better conceived and usually have more function per instruction than programs that utilize arrays. According to Mills (1986), independent estimates indicate that programs utilizing structured data flows have up to five times as much function per instruction than would be expected of programs utilizing arrays.

Incremental Development

Incremental development is used to help give designers intellectual control over the problem as well as maintain focus on the task at hand. By partitioning the development problem into manageable increments (usually less than 10,000 lines of code and less than eight staff months of effort), intellectual control over the development of complex systems can be established. Each increment defines a complete, user-executable system with added functionality over previous increments. Once an initial understanding of the requirements is achieved, the system is partitioned into increments based on criteria such as increment size, component reuse, and development team skills. While the requirements for unprecedented systems may not be entirely known, the portions that are known should be stable. Additional requirements can be brought under control and introduced at reasonable intervals. Increments may also be left open in certain aspects for later updates. Thus, further steps of requirements determination can be performed with each increment specification (SET, 1993).

Team Organization

Cleanroom projects involve three different types of teams: specification, development, and certification. Specification teams are responsible for preparing and maintaining the system specifications. Configuration management is the responsibility of the specification team.

Development teams design and build one or more of the software increments. The resulting source code is handed over to the certification team who compiles and tests the software. Development teams are also responsible for isolating and making any necessary changes to the increment. The number of development teams used on a project depends on the size of the system to be developed.

Each certification team prepares test cases for an increment. When the increment is submitted for certification, the team performs the testing and prepares the certification report. A certification team executes and tests the code but does not modify it in any way. Failures are reported to the appropriate development team for correction. As was the case with development teams, the number of certification teams depends on the size of the system to be developed.

Engineers can serve on more than one team. For instance, an engineer may work on the specification team and the development team. However, engineers are not normally members of the development and certification teams because of the principle of separation of development and testing (SET, 1993).

Box Structure Design

Cleanroom provides a rigorous basis for developing software by exploiting the fact that programs are rules for mathematical functions. The specification for a program must define a function that completely describes the behavior required for the software to fulfill its intended role. Finding and documenting this function is a specification task. Designing and implementing a correct procedure for the specified function is a development task.

The mathematical nature of software is leveraged in Cleanroom through the application of box structure mathematics to software specification and development. In fact, **the underlying mathematical foundations of box structures permit the scale-up of analysis and design to systems of arbitrary size**. Consequently, specifiers

and developers only need to work with three classes of functions: black boxes, state boxes, and clear boxes. Each of these box structures exhibits identical external behavior, but with an increasing degree of internal visibility. Figure 2.7 depicts the three box structures.

The technical details of box structure design are crucial for understanding the implications of this method. Details are provided in Appendix A. Important elements of the methodology will be highlighted below.

A black box provides an implementation-free, object-oriented description of software. This box structure only describes the software's external behavior in terms of a mathematical function that maps a stimulus history, S^*, to a response, R. Since the black box view excludes all details of internal structures and operations, it also provides a description of the user's view of system behavior.

A state box provides a data-oriented view that begins to define implementation details by modifying the black box to represent responses in terms of the current stimulus, S, and state data that contains the stimulus histories.

A clear box provides a process-oriented view that completes the implementation details by modifying the state box view to represent responses in terms of the current stimulus, state data, and invocations of lower level black boxes.

The effective use of box structure design methods for the development of systems is guided by the application of six basic box structure principles: referential transparency, transaction closure, state migration, common services, correct design trail, and efficient verification.

Referential Transparency. Each object is logically independent of the rest of the system and can be designed to satisfy a well defined "local" behavior specification.

Transaction Closure. The principle of transaction closure defines a systematic, iterative specification process to ensure that a sound and **complete** set of transactions is identified to achieve the required system behavior.

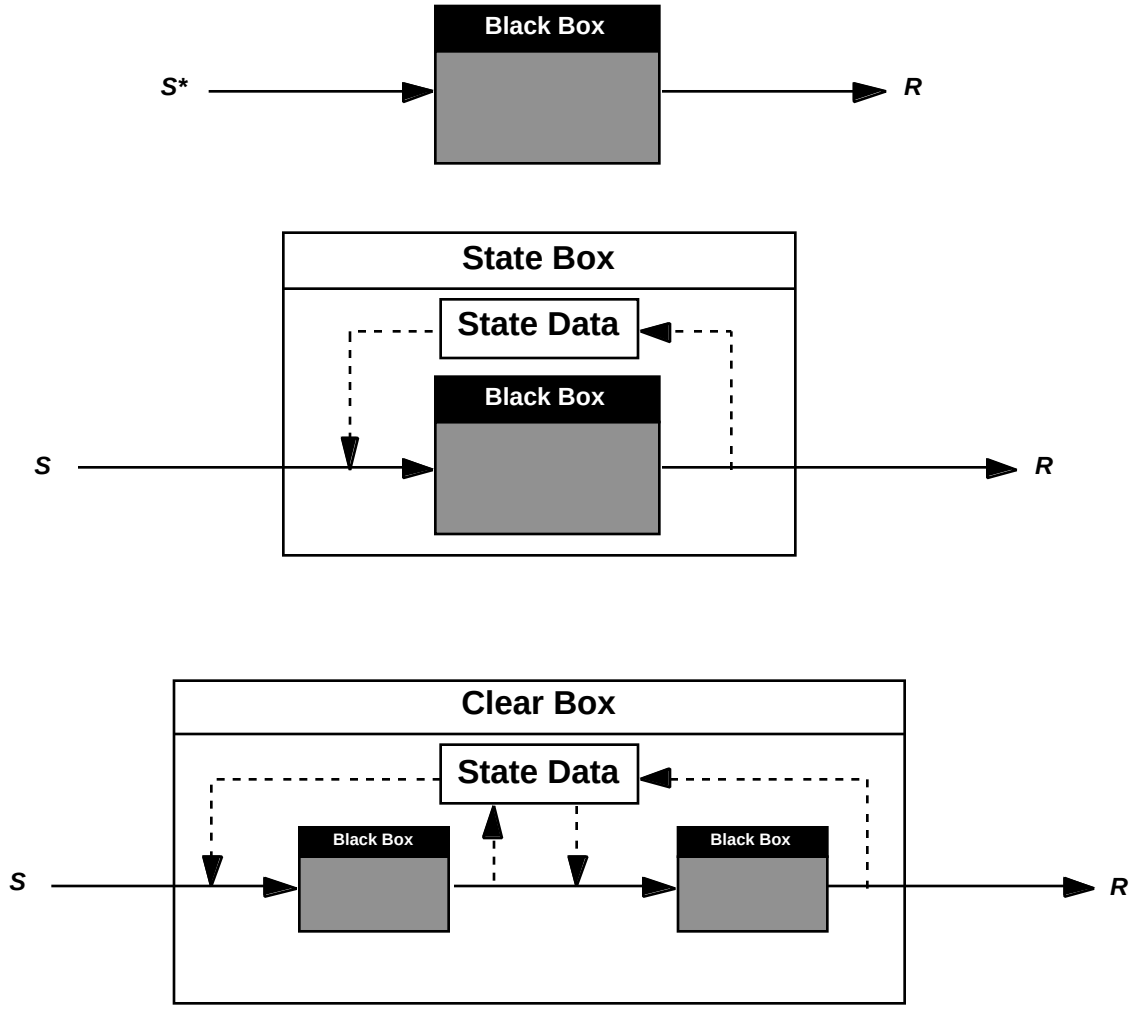
State Migration. State data is identified and stored in the data abstraction at the lowest level in the box structure hierarchy that includes all references to that data. The result is that state data can easily be transferred to the lowest feasible level.

Common Services. System parts with multiple uses are defined as common services for reusability. In the same way, predefined common services, such as database management systems and reuse objects, are incorporated into the design in a natural manner. The results are smaller systems and designs which accommodate reuse objects.

Correct Design Trail. It is important to insure consistency in the entire design trail when correcting an error.

Efficient Verification. It is only necessary to verify what is changed from one refinement to the next since all elements of the design are referentially transparent.

Figure 2.7: Box Structure Diagram



Designing software with box structures is performed in a top-down manner. Once the top-down design is completed, the clear boxes can be implemented in code. The software code is verified by demonstrating the equivalence of the program and the design represented by the clear box refinement. While system design proceeds in a top-down fashion, the implementation of the design is accomplished in a bottom-up fashion. Designing top-down and then coding bottom-up allows the developers to exploit fully the principle of common services during the design phase and generalize the common services as much as possible during the coding phase.

Functional Verification

In Cleanroom engineering, functional verification is used instead of unit debugging.[18] These functional verifications typically yield surprising improvements in design, even for the best software engineers. As a result, the developed software can be smaller and faster than previously thought possible, delivering more functionality per instruction. In addition, using functional verification allows quality to be designed into the software. According to Cobb and Mills (1990), functional verification leaves only two to five defects per thousand lines of code to be fixed in later phases of the life cycle whereas debugging leaves 10 to 30 defects per thousand lines of code. In contrast to functional verification, debugging attempts to test quality into the product. However, since more than 15 percent of the corrections merely introduce newer, deeper errors, testing quality into software is not possible (SET, 1993).[19]

Statistical Testing

Cleanroom engineering makes use of statistical usage testing to certify the reliability of the developed software in terms of its MTTF (Mean Time To Failure). The application of rigorous statistical theory allows both quality control of the software being developed and process control over the development of the software.[20]

---

[18] A more detailed discussion of functional verification practices is available in Appendix A.

[19] DeMarco (1982) contains an excellent analysis which demonstrates the validity of this point. Testing seems to be capable of eliminating half of the software defects. However, this factor of two improvement is overwhelmed by the extreme variability in the quality of software being produced today.

[20] A more detailed discussion of statistical testing practices is available in Appendix A.

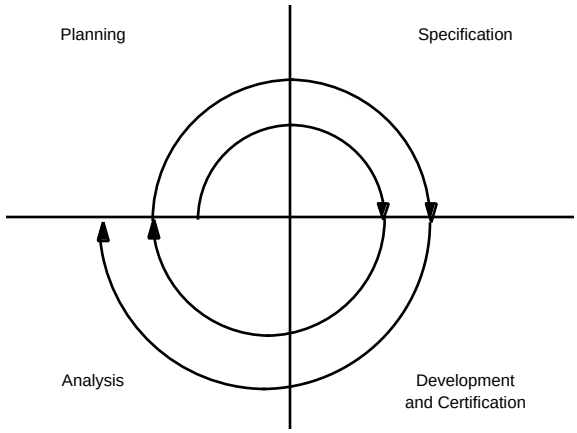
2.4.3 Cleanroom Engineering Process

Cleanroom projects follow a defined process which details the control structure between all processes and practices to be used in the project. Organizations which practice Cleanroom software engineering typically possess a set of defined processes where each defined process is intended for use with a different class of development project. Each of these defined processes, in turn, can be tailored to satisfy the specific needs of the project at hand.

Large development projects employ a spiral development process which partitions the problem into smaller, more manageable projects or spirals. Decomposing a large problem into smaller pieces reduces risk by establishing intermediate evaluation points. This allows projects to be redirected if necessary. Partitioning the development problem also enables the use of intermediate deliveries. This practice can be used to keep development efforts focused, elicit user feedback, and help assess the progress of the project. Figure 2.8 depicts the Cleanroom spiral development process.

Each spiral is divided into four quadrants or phases: planning, specification, development and certification, and analysis. The planning phase is where the project blueprint is devised. During this phase, the appropriate defined process model is chosen and tailored to the needs of the current development project. If an appropriate process model does not exist, a suitable process must be designed. Other planning activities include conducting risk analyses and setting objectives for each spiral (SET, 1993).

Figure 2.8: Cleanroom Engineering Spiral Development Process



During the specification phase, system specification development activities are performed. Typically, this involves close cooperation between the specification team and the customer and system users. Specification development tasks include analyzing the problem and solution domains. Problem domain analysis involves reverse engineering similar systems, developing the usage profile in the form of a Markov model, and formulating black box specification models. The formulation of black box functions is a critical task since it defines the functional behavior of the system to be developed. By defining the functional behavior of the system, the specification establishes *what* is to be done, not *how*. Solution domain analysis can involve such activities as reuse analysis and rapid prototyping. In Cleanroom engineering, the rapid prototyping process is similar to the standard Cleanroom process. The rapid prototyping process also includes planning, specification, development, and certification phases. However, once these phases have been completed, experiments can be conducted to acquire any additional information that may be necessary for specification development. While the specification team is responsible for prototype development and experimentation, development and certification teams are also typically involved in the process. When the prototyping effort ends, system modules that were developed and certified are placed in the project reuse repository and can be used by the development team to design and build the final product.

The construction plan for the spiral is also produced during the specification phase. Its purpose is to lay out a plan for the activities of the development and certification teams. The construction plan contains the actual modules and functions that are to be developed for each increment. Since each increment is integrated with the previous ones, the plan also defines the cumulative functionality that will need to be certified following the development of each increment.

The third phase of a spiral is the development and certification phase. Development and certification activities are performed in parallel. Development activities include designing each increment top-down with box structures, implementing the design in code, and verifying the correctness of the code through functional verification. Typically, design languages and automated code generation are used as much as possible to translate designs into code. During this time, the certification team formulates test cases. Once the development team delivers the source code for the current increment, it is integrated with

the previous increments and tested. If defects are detected, it is reported to the development team for correction. The corrected code is delivered to the certification team for re-evaluation.

Analysis is the final phase of the Cleanroom spiral. The results of the development cycle are analyzed in this phase. In addition, system demonstration and follow-up appraisals can be conducted (SET, 1993). The results of the work performed in this portion of the development cycle serve as inputs to the planning activities of the next spiral.

Each successive spiral builds on the work of the previous spirals. For instance, the specification for the overall system is developed in an iterative spiral. Each specification iteration further extends and/or refines the system specification.

2.4.4 Proven Benefits

Cleanroom is not a blue sky concept. The methodology is currently in use by many software engineering organizations across the United States and internationally as well. The implementation of Cleanroom engineering methods has consistently produced significant improvements in quality and productivity. A sample of the results of some Cleanroom projects is shown in Table 2.5.

Table 2.5: Sample of Cleanroom Results

Year	Project	Results
1987	Flight Control 33 KLOC (Jovial)	• Completed Ahead of schedule • <2.5 defects/KLOC before any execution • Defect-fix effort reduced by a factor of five
1988	Commercial product 80 KLOC (PL/I)	• Certification testng failure rate of 3.4 defects/KLOC • Deployment failures of 0.1 defects/KLOC • Productivity of 740 LOC/staff-month
1989	Satellite control 30 KLOC (Fortran)	• Certification testing failure rate of 3.3 defects/KLOC • 50% quality improvement • 80% productivity improvement • Productivity of 780 LOC/staff-month
1990	Research project 12 KLOC (Ada and ADL)	• Certified to 0.9978 reliability • Certification testing failure rate of 1.7 defects/KLOC

Source: Cobb and Mills (1990), Mills (1991).

A technology transfer effort implemented Cleanroom engineering processes and practices at a Picatinny Arsenal software engineering center and achieved impressive improvements immediately. The Picatinny software engineers improved their productivity by a factor of three on the very first increment on which the new methodology was used. The failure rate also displayed dramatic improvement. The failure rate for the first increment was only 0.24 failures per thousand lines of code (Sherer et al., 1994).

Even partial implementation of Cleanroom processes and practices seem to make substantial improvements. Head (1994) reported significant improvements from implementing just a few of the Cleanroom practices at Hewlett-Packard. A defect density of one defect per thousand lines of code was achieved on the first application of these practices to a project.

CHAPTER 3
Evaluating the New Methods

Now that we have described some of the fundamental ideas and practices behind a selected set of complex electronic system development methodologies, we must define a set of criteria by which to assess their utility for lean hardware/software development. Once a set of criteria and a rating scheme have been devised, the evaluations can be conducted to determine which methodology, if any, can be used for the lean development of complex electronic systems.

3.1 THE CRITERIA

To define a set of criteria that would characterize an ideal lean development methodology for electronic hardware and software, the characteristics of lean product development and high performance software engineering processes were considered as well as relevant system design issues. Successive rounds of distillation removed criteria that were thought to be redundant or nonessential for the purposes of this report. The final set of criteria is not a comprehensive checklist of everything necessary to define a lean hardware/software development methodology. It does, however, define a key set of process traits, which have been widely associated in other contexts with successful product development processes, and system design issues that should be encompassed by an ideal lean hardware/software development methodology. The final set of criteria is shown in Table 3.1.

Table 3.1: Ideal Cross-System Integration Methodology Criteria

Process Characteristics	• Defined • Configuration management • User involvement • Transparent • Tailorable • Rapid development cycle • Scalable methodology • Defect0free • Continuous improvement & lessons learned
Systems Design Issues	• Manufacturability • Supportability • Upgradability • Scalable architechture • Hardware/software design

The first nine criteria fall into the category of process characteristics. The rest of the criteria are system design issues that should be accounted for by an ideal lean hardware/software development methodology. The criteria are defined and the reasons behind their selection are discussed in the subsections that follow.

3.1.1 Process Characteristics

Defined

A hardware/software development methodology should follow a defined process. Experience has shown that developing hardware/software systems with an ad hoc approach usually results in poor project performance. Poor system performance and the all too familiar schedule and cost overruns are usually the results of a poorly defined process. Even if the project is successful, the lack of a defined process diminishes the chances that the success can be repeated (Boehm, 1976; Cusumano, 1991; Paulk et al., 1993a and 1993b).

Configuration Management

The system design configuration must be kept up-do-date so that designers can have the most current information at their disposal. This is a basic function for hardware/software system development (Paulk et al., 1993a and 1993b; Cusumano, 1991). Configuration

management activities include identifying the design configuration at certain points in time, systematically controlling changes to the design configuration, and maintaining the integrity and traceability of the configuration throughout the development cycle.

User Involvement

Users must be involved throughout the development process to ensure that the system meets their needs, which are often different from the stated requirements (SAF/AQK, 1992; IBM, 1990). A higher degree of user involvement enables the designers to gain a better understanding of the users' "true" requirements. This is particularly true for programs where man-machine interfaces are a factor.

An example of the utility of user involvement can be found in the discussion of GritTech's rapid development process located in Chapter 2. Without user involvement, the rapid development engineers would not have known to address the unstated requirement for a capability to call up video views of the launch vehicle and the launch pad. Adding this functionality to the system would have been significantly more problematic and costly to accomplish later in the development cycle.

The design of AH-64 Apache crew station provides another example of the utility of user involvement in the development process. Conforming to MIL-STD-704A, the Apache's electronic systems were interconnected with its two onboard generation systems so that a single generator could supply power for all the electronic systems in case one of the generators failed or was switched off. No one had anticipated that both generators could be switched off in-flight, which happened once inadvertently while flying nap-of-the-earth during the flight test phase. The pilot's partially rolled up sleeve caught both generator switches, switching off electrical power for all onboard systems except for the few items powered by battery. Only the skillful reactions of the crew averted a tragedy. Accordingly, the generator switches were subsequently redesigned to be lever-locked in the "on" position (Amer et al., 1992).

Transparent

The methodology should be easily understood by designers and managers alike. In addition, the process should allow development progress to be easily measured and tracked on a continuous basis. Transparency allows problems to be identified earlier in the

development cycle when they are not as costly to fix. A lack of transparency is a commonly cited problem for project management, particularly in the development of software-intensive systems (Paulk et al., 1993a and 1993b; SAF/AQK, 1992; Cusumano, 1991).

Tailorable

A single, defined process cannot be appropriate for all possible development projects. SDC's (System Development Corporation) experiments with a factory approach to software development in the mid-1970s, documented in Cusumano (1991), failed in part from the use of the standardized process on projects with widely varying needs. Predictably, the factory approach worked well on the types of development projects for which it was designed but produced disappointing results when applied to projects that were beyond the intended scope of the process (Cusumano, 1991). Attempting to define a single standard process to handle all situations typically results in a non-transparent, byzantine process that is unwieldy at best (Hammer and Champy, 1993). Thus, an ideal process should be tailorable to meet the particular needs of a project.

Rapid Development Cycle

A rapid development cycle, a basic element of lean or total quality product development processes (Womack et al., 1991; Ling, 1993; Clausing, 1994), is important for a number of reasons. First, a rapid development cycle places new capabilities in the hands of the warrior faster. Moreover, rapid development cycles are needed to ensure that systems are state of the art when fielded. Currently, development cycles are so lengthy that some systems can be obsolete by the time they are fielded. Rapid development cycles can also facilitate timely upgrading of current systems to keep their capabilities at or near the rapidly improving state of the art in hardware and software technology.

Scalable Methodology

An ideal methodology would be efficient and effective for both relatively simple and extremely complex hardware/software system development. For instance, there are processes that appear extremely efficient and effective for developing small systems. However, when applied to a larger development project, information flows which were

crucial to the success of the process for smaller projects can break down—particularly the informal channels, primarily due to the larger number of people involved.[1] Similarly, design methods that work well for small problems can be swamped by the number of constraints and variables involved in solving larger, more complex design problems. Moreover, a scalable methodology is needed to keep pace with the increasing complexity of electronic system development problems.

Defect-Free

An ideal methodology should aim to "get it right the first time" and develop a high-quality, defect-free finished product. Unfortunately, while this is usually the goal, the finished product has not always lived up to these expectations in the past. In spite of the myriad process and product standards that have been imposed on defense contractors, systems have been deployed with known defects. For instance, CAFMS (Computer Assisted Force Management System), a computer system used to develop air tasking orders, was deployed to the Gulf with several known software defects. In all, there were 81 major software changes made during Desert Shield and Desert Storm. While some were the result of the growing scale of operations, many changes were made to fix known problems that had been brought to the field and others that appeared during Desert Shield (Hyde et al., 1992).

Defect prevention practices must be a part of a methodology to fully satisfy this criterion. In the absence of defect prevention practices, methodologies which aim to produce a defect-free product can only partially satisfy this criterion since they tend to rely on substantial amounts of testing to detect defects for subsequent removal rather than preventing their introduction. Defect prevention is significantly more effective for assuring product quality than testing (Head, 1994).

Continuous Improvement & Lessons Learned

Continuous improvement, an essential element of total quality development (Clausing, 1994), should be part of an ideal cross-system integration methodology. Continuous improvement and documenting lessons learned are critical for fostering organizational

[1] This sort of problem is well documented in the literature by Lawrence and Lorsch (1967), Lawrence et al. (1976), and Lawrence and Lorsch (1986).

learning.[2] While defect prevention activities are covered by the defect-free criterion, there are certainly numerous other process aspects that could be improved. For instance, it may be possible to accelerate the pace of development activities by rearranging certain elements in the workflow. In addition, technical and process-related lessons learned during each project should be documented so that they are not forgotten. Subsequent projects should benefit from this knowledge, not just the individuals who participated. Toyota's "lessons learned" books provide a good example of this (Ward et al., 1994). For example, one book is comprised of lessons learned in fender design and contains approximately 60-72 different key ranges of specifications that would ensure the manufacturability of fender designs. These lessons learned books, which exist for every body part, allow Toyota designers to ensure their designs are manufacturable from the start (Ward et al., 1994).

3.1.2 System Design Issues

Manufacturability

An ideal methodology should produce a design that is manufacturable. Manufacturing processes should be able to produce the integrated system and all its components in an efficient, affordable manner. Since the design of a product can, by some estimates, determine as much as 70 to 80 percent of manufacturing productivity, it is critical to address manufacturability issues during the design phase (Suh, 1990). Design for manufacturability is another basic element of lean product development (Womack et al., 1991; Ling, 1993; Clausing, 1994).

In the past, manufacturability problems have required costly redesign efforts to correct. Printed wiring boards, used extensively throughout the LANTIRN system (Low-Altitude Navigation Targeting Infrared for Night), had to be redesigned because the hand-made boards tended to fracture when machined and could not pass the required acceptance tests (Bodilly, 1993b). Manufacturability problems with the AMRAAM (Advanced Medium-Range Air-to-Air Missile) missile led to the establishment of the $330 million AMRAAM Producibility Enhancement Program (Mayer, 1993). Manufacturability problems and a failed aluminum casting process forced a very costly redesign of the

[2] The subjects of learning organizations and organizational learning are well documented by Senge (1990) and Argyris (1991 and 1993).

stealthy TSSAM (Tri-Service Standoff Attack Missile), and have provided ammunition to those calling for the cancellation of the program (Morrocco, 1993; Fulghum, 1994e; Fulghum and Morrocco, 1994).

Supportability

Systems should also be easy to maintain in the field. Historically, maintenance costs of hardware/software systems exceed that of original system development. If operations and maintenance costs are included in a software life cycle cost breakdown, they account for 67 percent of the total life cycle cost (Cusumano, 1991).[3] From the hardware perspective, maintenance tests and diagnostics should be easy to perform and crucial areas should be as easily accessible as possible. Similarly, software components should be easy to access, test, and replace.

Supportability has become an increasingly important consideration in the development of weapon systems and their subsystems. Both the Air Force's Advanced Tactical Fighter and the Army's Light Helicopter competitions placed great emphasis on supportability issues (Nordwall, 1993; Bond, 1991; Kandebo, 1991). Supportability problems with Pratt & Whitney's F100-PW-100 and -200 engines—used in the Air Force's F-15 and F-16 fighters, respectively—provided a major part of the motivation for the initiation of the Alternative Fighter Engine (AFE) competition. In addition to addressing a stall-stagnation problem, the AFE competition addressed the F100's extremely short lifetime—the period of time between depot overhauls—and its high maintenance requirements which drove up operating costs (Camm, 1993a).

Upgradability

Considering the typically lengthy service lives of many defense systems[4] and the rapid pace of improvement in hardware and software technology, an ideal methodology should yield an upgradable product. This would allow a fielded system to keep pace with

---

[3] This frequently reflects the costs imposed by a defective product and a flawed process that developed it. Activities in this portion of the life cycle are needed to fix errors that escaped detection during development and to give it the functionality that users really wanted.

[4] For example, the F-4 has been in service since the 1960s, and the F-15 and F-16 fighters have been in service since the 1970s.

technology over its service lifetime with greater ease and less expense than is required by current approaches.

Scalable Architecture

An ideal methodology should produce a system architecture that can be extended to fulfill the evolving needs of the user over the service life of the system. Typically, systems will be called upon to perform functions that were not part of the original design requirements or are much larger in scope. An ideal methodology should yield an architecture that can scale up to meet the new challenges. Recent events point to the need for scalable architectures. CAFMS was not designed to handle the number of air bases or the size of the ATO required in Desert Storm. Fortunately, we had the luxury of five months with which to enhance the system with additional processors and storage devices, some of them newly acquired, and with software changes engineered in the desert (Hyde et al., 1992).

Current efforts to add data fusion capabilities and other enhancements to AWACS presents another example where non-scalable architectures are causing problems. Although AWACS performed brilliantly during the Gulf War, the demands of the scope and pace of allied operations were beginning to push the system to the limits of its capabilities (Lenorovitz, 1992a). Unfortunately, the aging centralized mainframe computer architecture makes it more difficult to enhance system capabilities or add the latest in distributed computer hardware. For instance, the 1500 lb. electronics unit that drives the AWACS displays is not programmable which makes it difficult to devise and implement better ways to display data with software modifications. Making similar enhancement or upgrades for JSTARS aircraft is much easier because of its distributed, open architecture which utilizes high-speed commercial off-the-shelf engineering workstations (Hughes, 1994a; 1994b).

HW/SW Codesign

An ideal methodology should allow for interaction and tradeoffs between hardware and software development to balance flexibility, performance, and cost within each separate system. Traditionally, system functionality is partitioned between hardware and software early on in the development process, and subsequent development activities are pursued with little or no interaction between the groups responsible for the implementations. The

result is often unsatisfactory performance. Codesign allows more interaction and tradeoffs between hardware and software implementations. This allows a more thorough exploration of the "design space" which yields an end product that provides a better balance of single system flexibility, performance, and cost.

3.2 THE RATING SCHEME

The rating scheme chosen to evaluate the different methodologies is fairly straightforward. For each criterion, three different ratings are possible: satisfies, partially satisfies, and does not satisfy.

3.3 THE EVALUATIONS

This section contains the evaluations of each of the methodologies described in the previous chapter. The results of the evaluations are shown in two matrices corresponding to the two categories of criteria. Table 3.2 shows how the methodologies rated according to the process characteristics criteria. Table 3.3 shows how the methodologies rated according to the system design issues criteria. The reasons behind the ratings are provided in the subsections that follow.

3.3.1 DC-X Rapid Development

The reasons behind the ratings of this methodology are discussed in this section on a criterion-by-criterion basis.

Defined	Satisfies. This process follows a defined process.
Configuration Management	Satisfies. The process has mechanisms for configuration management.
User Involvement	Satisfies. The process allows simulations, which can be tested by users, to be produced in parallel with the flight software.

Table 3.2: Process Criteria Matrix

	DC-X Rapid Development	GritTech Rapid Development	Ptolemy HW/SW Codesign	RASSP	Cleanroom Engineering
Defined	●	●	●	●	●
Configuration Management	●	●	●	●	●
User Involvement	●	●	○	●	●
Transparent	●	●	●	●	●
Tailorable	●	●	●	●	●
Rapid Development Cycle	●	●	●	●	●
Scalable Methodology	○	○	◐	●	●
Defect-Free	◐	◐	◐	◐	●
Continuous Improvement & Lessons Learned	○	◐	○	●	●

○ Satisfies ◐ Patially Satisfies ○ Does Not Satisfies

Transparent Satisfies. Designers and managers have a good sense of the status of the project and the rate of progress.

Tailorable Satisfies. The process is tailorable to the needs of the project.

Rapid Development Cycle Satisfies. The process is characterized by a rapid development cycle and fully utilizes rapid prototyping.

Scalable Methodology Does Not Satisfy. The methodology is not scalable. The process is highly dependent upon a toolset which seems to "break" when applied to large problems.

Table 3.3: System Design Criteria Matrix

	DC-X Rapid Development	GritTech Rapid Development	Ptolemy HW/SW Codesign	RASSP	Cleanroom Engineering
Manufacturability	○	◐	●	●	○
Supportability	○	◐	○	◐	◐
Upgradability	○	○	○	◐	◐
Scalable Architechture	○	○	◐	◐	◐
HW/SW Codesign	○	○	◐	◐	○

○ Satisfies ◐ Patially Satisfies ○ Does Not Satisfies

Defect-Free Partially Satisfies. The process seeks to develop a defect-free product, but does not practice defect prevention.

Continuous Improvement & Lessons Learned Does Not Satisfy. This process has no provisions for continuous improvement or documenting lessons learned.

The DC-X rapid development process does not satisfy any of the criteria pertaining to the system design issues since there is no provision in the process for the consideration of these issues.

3.3.2 GritTech Rapid Development Evaluation

The reasons behind the ratings of GritTech's rapid development methodology are discussed in this section on a criterion-by-criterion basis.

Defined Satisfies. The GritTech rapid development group has a defined process.

Configuration Management Satisfies. The process has mechanisms for configuration management.

User Involvement	Satisfies. User involvement is integral to the success of the software development approach practiced by GritTech.
Transparent	Satisfies. Designers and managers have a good sense of the status of the project and the rate of progress.
Tailorable	Satisfies. The GritTech rapid development process is highly. Designers have great freedom over tailoring the process to the needs of the project in question.
Rapid Development Cycle	Satisfies. The process is characterized by a rapid development cycle.
Scalable Methodology	Does Not Satisfy. The methodology relies heavily on informal communication among team members. Since the information flows would break down for large projects, the methodology is not scalable.
Defect-Free	Partially Satisfies. The process seeks to develop a defect-free product, but does not practice defect prevention.
Continuous Improvement & Lessons Learned	Partially Satisfies. Since activities associated with continuous improvement and documenting lessons learned are conducted at the discretion of the development team, they are not always performed.
Manufacturability	Partially Satisfies. The process can include this consideration, but it can be (and has been) tailored out of the process in the interests of speeding up the development cycle.
Supportability	Partially Satisfies. Again, this consideration can be a part of the process, but it can be tailored out of the process.

The GritTech rapid development process does not satisfy any of the remaining system design criteria since the methodology has no provision for the consideration of these issues.

3.3.3 Ptolemy Hardware/Software Codesign Evaluation

The reasons behind the ratings of a Ptolemy-supported hardware/software codesign methodology are discussed in this section on a criterion-by-criterion basis.

Defined	Satisfies. This process follows a defined process.
Configuration Management	Satisfies. The process has mechanisms for configuration management.
User Involvement	Does Not Satisfy. HW/SW codesign does not involve the user in the design process, partially due to the fact that it relies on other methods to perform requirements capture.
Transparent	Satisfies. Progress is easy to monitor with this methodology.
Tailorable	Satisfies. The process is tailorable depending on the type of codesign problem involved. Different types of problem require the application of different design methodologies.
Rapid Development Cycle	Satisfies. The process is characterized by a rapid development cycle.
Scalable Methodology	Partially Satisfies. The methodology is scalable for a large range of design problems, but it is not capable of including the use of high-level languages in the codesign process.
Defect-Free	Partially Satisfies. The process seeks to develop a defect-free product, but does not practice defect prevention.
Continuous Improvement & Lessons Learned	Does Not Satisfy. This methodology does not currently include activities for continuous improvement and documenting lessons learned.

Manufacturability	Partially Satisfies. Depending on the specific development project, estimates of the cost to manufacture a design are considered.
Supportability	Does Not Satisfy. Supportability is not currently considered by Ptolemy-supported codesign.
Upgradability	Does Not Satisfy. Upgradability is not currently considered by Ptolemy-supported HW/SW codesign.
Scalable Architecture	Partially Satisfies. Depending on the specific development project, scalable architectures can be developed.
HW/SW Codesign	Partially Satisfies. The design method is limited to tradeoffs between hardware and low-level software. High-level software is not included in the codesign process.

3.3.4 RASSP Evaluation

RASSP was evaluated even though its work is still in its infancy. This evaluation will help see what future improvements (if any) there may be. The reasons behind the ratings of RASSP development methodology are discussed in this section on a criterion-by-criterion basis.

Defined	Satisfies. This process follows a defined process.
Configuration Management	Satisfies. The process has mechanisms for configuration management.
User Involvement	Satisfies. Users become a part of the development process in the model year design concept.
Transparent	Satisfies. Virtual prototyping and the workflow management features of the RASSP development environment provide a high level of transparency in the development process.

Tailorable	Satisfies. The RASSP process will be tailorable to the needs of the project at hand. Some of the tailoring will come naturally as a result of the design methodologies required to solve the current design problem.
Rapid Development Cycle	Satisfies. The process is characterized by a rapid development cycle.
Scalable Methodology	Satisfies. The methodology is applicable to a broad range of digital systems design.
Defect-Free	Partially Satisfies. The process seeks to develop a defect-free product, but does not practice defect prevention.
Continuous Improvement & Lessons Learned	Satisfies. The methodology utilizes continuous improvement and documents the lessons learned from each model year design.
Manufacturability	Satisfies. This is a design consideration of the RASSP design methodology.
Supportability	Partially Satisfies. While RASSP does satisfy this criterion for embedded digital systems, it does not ensure supportability for the range of systems that an ideal lean hardware/software development methodology would need to address.
Upgradability	Partially Satisfies. The model year concept of design emphasizes the development of upgradable designs. However, while RASSP does satisfy this criterion for embedded digital systems, it does not ensure upgradability for the range of systems that an ideal hardware/software development methodology would need to address.
Scalable Architecture	Partially Satisfies. While RASSP does satisfy this criterion for embedded digital systems, it does not ensure scalable

	architectures for the range of systems that an ideal hardware/software development methodology would need to address.
HW/SW Codesign	Partially Satisfies. While RASSP does satisfy this criterion for embedded digital systems, the design method is limited to tradeoffs between hardware and low-level software. High-level software is not included in the codesign process.

3.3.5 Cleanroom Engineering Evaluation

The reasons behind the ratings of the Cleanroom engineering methodology are discussed in this section on a criterion-by-criterion basis.

Defined	Satisfies. This process follows a defined process.
Configuration Management	Satisfies. The process has mechanisms for configuration management, which is performed by the specification team.
User Involvement	Satisfies. In the least, users are deeply involved in the specification phase of each project spiral.
Transparent	Satisfies. The detailed specifications and top-down design with box structures provide a good sense of project status and progress. Metrics tracked during the project also usually provide an accurate gauge of progress.
Tailorable	Satisfies. Project planning tasks include tailoring the process to the needs of the current project.
Rapid Development Cycle	Satisfies. The process is characterized by high productivity and a rapid development cycle.
Scalable Methodology	Satisfies. Box structure design and functional verification practices are scalable.
Defect-Free	Satisfies. This methodology involves defect prevention.

Continuous Improvement & Lessons Learned	Satisfies. Activities related to continuous improvement and documenting lessons learned are performed in the analysis phase of each project spiral as well as at the end of the development project.
Manufacturability	Does Not Satisfy. Manufacturability is not currently considered in Cleanroom engineering.
Supportability	Partially Satisfies. Referential transparency and box structure design produce a supportable software design. However, Cleanroom does not ensure the design of supportable hardware.
Upgradability	Partially Satisfies. Referential transparency and box structure design produce an upgradable software design. However, Cleanroom does not ensure the design of upgradable hardware.
Scalable Architecture	Partially Satisfies. Referential transparency and box structure design produce a software design that can be easily extended and enhanced. However, Cleanroom does not ensure the design of scalable hardware architectures.
HW/SW Codesign	Does Not Satisfy. Hardware/software codesign is not part of Cleanroom engineering practices.

3.4 CONCLUSIONS

In terms of process characteristics, only Cleanroom engineering satisfied all the criteria. RASSP also scored very well, but fell just short on the defect-free criterion. The other methodologies all displayed shortcomings on multiple counts (see Table 3.2). Thus, it appears from this standpoint that methodologies exist that have general process characteristics that could provide a good foundation for a lean hardware/software system development methodology.

Unfortunately, a glance at the system design criteria matrix in Table 3.3 indicates that none of these methodologies are directly applicable as a lean development methodology in their current form. Very few of the system design issues are addressed by any of these methodologies. Moreover, when these issues are addressed at all, the methodologies usually only partially satisfy the criteria.

Based upon these evaluations, we can make the following conclusions:

- There are methodologies that possess general process characteristics that would provide a good basis for a lean hardware/software development methodology.
- Very few of the system design issues are addressed by these methodologies.
- None of the complex electronic system development methodologies in their current forms can meet the need for a lean hardware/software development methodology.

CHAPTER 4

Conclusions

Electronic hardware and software are key determinants of the performance of defense systems. Current challenges include devising a development methodology capable of scaling to cope with increasing complexity and producing a high-quality product that is affordable, upgradable, evolvable, and maintainable.

Several methodologies for developing complex electronic systems were evaluated in the previous chapter to determine if any could be employed directly as a lean hardware/software development methodology. The results indicated that while there are development methodologies that possess the process characteristics of an ideal hardware/software development methodology, very few of the important system design issues are addressed. Thus, none of the complex electronic system development methodologies is capable in its current state of adequately addressing all the challenges associated with the development of modern complex electronic systems. This finding has major implications.

4.1 TACKLING THE HARDWARE/SOFTWARE DEVELOPMENT CHALLENGE

While none of the methodologies evaluated in the previous chapter can adequately serve as a lean development methodology for complex electronic systems, the process criteria matrix shown in Table 3.2 indicates that a some of them exhibit process characteristics that

could provide a good foundation for an effective hardware/software development methodology. Specifically, this foundation could be provided by Cleanroom engineering, the RASSP design methodology, or a combination of the two methodologies.

Cleanroom engineering fully satisfied all the criteria related to process characteristics. However, it only partially satisfied the system design criteria of supportability, upgradability, and scalable architecture since it does not adequately address these aspects of hardware design. The strengths of Cleanroom engineering include a scalable design methodology that exploits the benefits of common services, referential transparency, functional verification, and statistical testing to produce nearly defect-free software. The originators of Cleanroom emphasize that it is applicable to system engineering as well as software engineering. Using box structure design, the system can be designed in a top-down manner. Once the design is completed, the clear boxes can be implemented in hardware or software. Since Cleanroom engineering is practiced primarily by the software engineering community, the process of implementing clear boxes in software is well understood. The process of implementing clear boxes in electronic hardware is not as mature. Cleanroom engineering does not adequately address digital hardware design issues, nor is it equipped to deal with tradeoffs between hardware and software. For instance, while Cleanroom engineering ensures the development of an upgradable software design, it does not ensure the development of upgradable electronic hardware.

The RASSP design methodology fully satisfied all but one of the criteria related to process characteristics. Moreover, it addressed more system design criteria—manufacturability, supportability, upgradability, scalable architecture, and hardware/software codesign—than any other methodology evaluated. It fully satisfied the manufacturability criterion, but only partially satisfied the others since the program is focused on the domain of embedded digital signal processors and may not provide an adequate scope to form a solid foundation for an effective cross-system integration methodology. In addition, since the methodology is still under development, it is not certain that the methodology—especially its system engineering aspects—will be as scalable as hoped.

The third alternative would involve a combination of the Cleanroom and RASSP approaches. This option, suggested by the results of the system design criteria evaluation discussed in Chapter 3, combines the strengths of the two methodologies while addressing the shortcomings of each. A synergistic combination of aspects of Cleanroom engineering and the RASSP design approach could provide a solid foundation for a lean development methodology for electronic hardware/software systems. Cleanroom offers a rigorous approach to system engineering and software development, and RASSP provides design methodologies and tools for hardware/software codesign of embedded digital systems.

Developing a methodology that synergistically combines different aspects of Cleanroom engineering and the RASSP design methodology would not be a trivial undertaking. Many issues would have to be resolved to accomplish this fusion successfully. Some of these issues include:

- Can the Cleanroom principle of separation of development and testing be reconciled with the RASSP approach of rapid prototyping and simulation of hardware/software systems?
- Can the RASSP capabilities of designing hardware that is manufacturable, supportable, upgradable, and scalable be integrated into the Cleanroom process of refining black boxes into clear boxes?
- Are the Cleanroom and RASSP approaches to ensuring supportability, upgradability, and scalability compatible with each other?

Assuming that a synergistic combination of Cleanroom and RASSP methodologies is feasible, a next step would be to develop an integrated system engineering environment to support hardware/software development activities.

4.2 BENEFITS OF LEAN HARDWARE/SOFTWARE DEVELOPMENT

Lean hardware/software development would have several benefits. First, development times could be significantly reduced. Use of fast cycle development methodologies such as Cleanroom engineering, the RASSP design methodology, or the

proposed combination of Cleanroom and RASSP should dramatically cut development times while producing high-quality products that are scalable, upgradable, and supportable. Accelerating the development of these systems through the use of such methodologies would also place needed capabilities in the hands of the warfighters earlier. Moreover, the fielded systems would be much closer to the state-of-the-art at the time of deployment. Under current development and acquisition practices, the performance of commercially-available systems can surpass the projected performance of a defense system before its development is completed.

Second, lean development as envisioned in this report would allow new and existing systems to keep pace with the rate of technological advance. Obsolescence is especially a concern in electronics and software because of the rapid rate of performance improvement of these technologies. Since the products of these development projects would be upgradable and scalable, product capabilities would be much easier to enhance and extend. Lean hardware/software development would also facilitate the application of the model-year design concept, which would further enhance the ability to keep pace with technological advance.

The higher productivity of a lean development methodology would translate into lower development costs, resulting in a more affordable final product once the decoupled subsystems are integrated. In addition, since the systems would be upgradable, scalable, and supportable, life cycle costs would also be significantly reduced (Richards, 1994). Further cost savings could be generated through reuse of software and hardware designs and off-the-shelf components. Moreover, applying the principle of common services during the system design process would facilitate substantial cost savings through the identification and exploitation of economies of scope.

In addition to these benefits, the fast cycle development approach offers the benefits of enhanced robustness to budgetary instability and enhanced risk reduction. The primary vehicle for both of these benefits is speed. Projects that can be completed rapidly are less likely to experience the effects of budgetary instability since there would be fewer opportunities for budgetary changes during development. Maximum robustness would be achieved by projects that could be completed within a single budget cycle. Risk would be reduced in two ways. First, rather than conducting numerous risk assessments and

analyses, a real system or prototype would be rapidly developed which would provide more accurate estimates of the risks involved based on actual experience and data. In addition, producing real hardware and software provides a more tangible result than the traditional risk reduction approach of extensive analyses, documentation, and milestone reviews. With all other factors equal, a project that can demonstrate working hardware or software is less likely to experience budget cutbacks than a project that only has reams of analyses and documentation to show for its money. Thus, the benefits of lean development of complex electronic systems would include improving robustness to budgetary instability and reducing development risks.

To summarize, the lean development of complex electronic systems as envisioned in this report would provide the following benefits:

- Reuse of hardware and software designs
- Rapid development of needed capabilities
- State-of-the-art operational systems
- Greater affordability
- Enhanced robustness to budgetary instability
- Reduced development risk

4.3 CONCLUSIONS AND RECOMMENDATIONS

This report is the start of a study of how to address the challenge of devising a development methodology capable of scaling to cope with increasing complexity and producing a high-quality product that is affordable, upgradable, evolvable, and maintainable. This problem is very complex, but there are concepts and methodologies—some of which are unconventional—that point the way to a solution.

4.3.1 Conclusions

- The effective and efficient development of electronics is critical to future U.S. military capabilities.

- To address the challenges inherent in the development of defense electronics, a lean hardware/software development methodology is needed.
- Although there are complex electronic system development methods that meet some of the criteria of a good hardware/software development methodology, none of the methods studied can be readily applied in its current form as a lean methodology for developing electronic systems.
- Cleanroom engineering and the RASSP design methodology rated the best in terms of process characteristics, but satisfy only a few of the system design criteria.
- Cleanroom engineering and the RASSP design methodology could be combined to form a foundation for a lean development methodology.

4.3.2 Recommendations

- Combine Cleanroom engineering and RASSP design practices synergistically to form a core from which to develop a lean hardware/software development methodology. Cleanroom engineering offers a rigorous methodology for system and software engineering that can scale-up to permit the design of systems of arbitrary size. The RASSP design methodology offers a scalable approach to designing embedded hardware/software systems.
- Apply fast cycle methods including Cleanroom engineering, the RASSP design methodology, and/or the proposed combination of Cleanroom and RASSP for rapid development of electronic systems. The benefits would be systems which are state-of-the-art at the time of deployment, upgradable, scalable, supportable, and more affordable. Needed capabilities are fielded faster. Rapid development also provides for improved robustness to budgetary instability and risk reduction.
- Specify functional requirements, not implementations (i.e., specify black box behaviors of subsystems). This provides designers with maximum flexibility in devising a solution to provide the required functionality.

- Develop an integrated design environment to support the lean development of hardware/software systems.

4.4 RECOMMENDATIONS FOR FURTHER STUDY

In the course of researching this report, a number of interesting possibilities for further study were identified. Some of these topics are important subjects related to lean development that were beyond the scope of this report. Other suggestions include further study of some of the methodologies that were selected for evaluation in this report. Some topics are interesting ideas that were encountered during the research process that may have potential for application in the aerospace industry.

4.4.1 Further Study of Selected Methodologies

While none of the methodologies are directly applicable as a lean hardware/software development methodology, the Lean Aircraft Initiative would benefit from knowledge gained from the further study of some of these methods. The following is a list of recommended actions for this purpose:

- Maintain contact with the ARPA/Tri-Service RASSP program. Draw upon the findings and the development work being conducted under its auspices. Monitor the progress of benchmarking activities and the evolution of the design process. At minimum, the Lean Aircraft Initiative would benefit from increased insight into the benefits and problems associated with integrated electronic design environments and collaborative design.

- Maintain contact with the ARPA STARS (Software Technology for Adaptable, Reliable Systems) program. Currently, this ARPA program is conducting multiple technology and process demonstration projects, some of which involve the Cleanroom engineering methodology described in this report. The Initiative would benefit from the insights generated by these demonstration projects as well as the wealth of software development data that will be produced and collected for analysis.

- Conduct an F-22 avionics development case study. It appears from several phone interviews conducted with former and current members of the avionics integrated product team that there were a number of substantial improvements over past programs in the development of avionics and software for the F-22. Among the innovations is a systems/software engineering environment that allows the designer to simulate the avionics system from high-level Ada code down to gate-level hardware functionality. Preliminary research indicates that a derivative of hardware/software codesign has been used. Other interesting aspects include the use of common hardware modules and the application of data fusion to integrate information and display it to the pilot in a more understandable form rather than requiring integration to take place "between the headphones".

4.4.2 Other Interesting Topics

New Development Challenges and Benefits of Better Integration

There are essentially two classes of problems that are important to the overall performance of a theater system:

- Single system development problems
- Cross-system integration problems

Single system development problems associated with the development of hardware/software systems are well documented and are the subject of numerous studies and research programs. However, while the lion's share of development energies and system integration activities are focused on optimizing the performance of individual systems or platforms (Rich and Dews, 1986), **effective integration across these systems is critical to the performance of the overall theater system**. In fact, inadequate integration is just as, if not more, important to theater system performance and amounts to self-inflicted degradation of military capabilities. Thus, **the new development challenge is to ensure effective integration across systems through rapid cross-system integration while producing a theater system**

that is flexible, rapidly deployable, upgradable, evolvable, and maintainable.

Effective cross-system integration holds great promise for increasing U.S. military capabilities. The following list provides a sample of the possible **benefits**.

1. More information could be transmitted by secure data link rather than voice communications which are easier to intercept.
2. Effective integration of precision-guided munitions and C4I systems could allow platforms (e.g., fighter aircraft) to operate with greater stealth since the platform would not have to use its own emitters to guide the munition to its target.[1]
3. Better integration can enable in-flight retargeting or rerouting of munitions, such as cruise missiles, to minimize collateral damage and ensure that primary targets are struck.
4. Platforms could employ the best munitions for their missions.[2]
5. U.S. forces could operate more seamlessly with forces from other services and different countries.
6. Our forces could deploy rapidly with a ready information infrastructure.
7. "Buddy system" tactics could be more easily employed since data and information from a particular onboard system could be shared with other platforms that were not similarly equipped.
8. Intelligence could be placed in the hands of the warfighters in a timely and efficient manner.[3]
9. Tactical and strategic situational awareness would be significantly enhanced.[4]

---

[1] Coarse guidance could be provided by systems such as JSTARS, and terminal guidance could be performed by the munition's on-board systems.

[2] This would significantly increase the flexibility and robustness of the capability of multi-role platforms.

[3] For instance, the most recent satellite imagery of a target area could be provided to pilots enroute to the objective.

[4] Fighter pilots could use AWACS- and JSTARS-supplied data as well as its own sensor data to provide a more complete tactical picture of the space around them and enable better discrimination between friend and foe. Data fusion could be used to display the information in a more intuitive manner. Theater commanders could be provided a "God's eye view" of all movements of enemy and friendly forces on or above the battlefield.

10. Effective integration would enable more efficient planning and dissemination of orders and plans such as the Air Tasking Order.[5]

Some of the benefits of effective cross-system integration are already being discovered by some currrent programs, including the F-16 HTS (HARM Targeting System) and JTIDS (Joint Tactical Information Distribution System). The performance of newly fielded AN/ASQ-213 HTS-equipped, F-16 Block 50/52D aircraft in the SEAD role has exceeded prior expectations to such a degree that the USAF believes the future development of a radar-killing version of the F-15 will be unnecessary. Air Combat Command is particularly pleased with the ability of the HTS to interact with off-board sensors such as Rivet Joint.[6] Although the HTS was originally intended as an interim system, planned upgrades should make the F-16 HTS system as capable as the radar-killing F-15 at a far lower cost (Morrocco and Fulghum, 1994).

JTIDS allows pilots to receive and view AWACS data on a color multifunction display in the cockpit (Morrocco, 1994). JTIDS allows a pilot to see the whole battlefield. For instance, instead of being limited to an F-15 radar's 120-degree forward field of view, a JTIDS-equipped F-15C provides its pilot with the ability to see all around his aircraft, including the locations of enemy surface and air threats (Fulghum, 1993e). With the JTIDS-enhanced situational awareness, the 390th Fighter Squadron, the only F-15C unit currently equipped with JTIDS, expects to be able to adopt the line abreast formation instead of flying in trail (Fulghum, 1993e). This allows forward firepower to be maximized and should result in greater lethality and fewer U.S. and Coalition losses.

Set-Based Design

As documented by Ward et al. (1994), Toyota's product development process, which is seemingly incongruent with widely accepted models of concurrent engineering, provides a second Toyota paradox. Contrary to conventional wisdom on concurrent engineering, delaying decisions and making many prototypes can make better cars faster and cheaper

---

[5] This would enable faster planning cycles and, hence, a faster pace of operations.

[6] The F-16 HTS system is considered proof of concept that with additional refinement, a single-seat aircraft can perform the SEAD role almost as well as a two-seat design. In the future, the electronic warfare officer could be on board an AWACS or Rivet Joint aircraft instead of in the cockpit of a Wild Weasel aircraft (Morrocco and Fulghum, 1994).

(Ward et al., 1994). Toyota's multidisciplinary development teams are neither co-located nor dedicated to a single project. Instead of trying to freeze specifications as rapidly as possible, Toyota engineers and managers deliberately delay decisions and provide deliberately ambiguous information to their suppliers. Moreover, instead of seeking to minimize the number of prototypes, Toyota and its suppliers produce a seemingly excessive number of prototypes (Ward et al., 1994).

The key to understanding this development approach is "set-based design". While more traditional processes utilize an iterative "point-to-point" design approach, in which the state of the design moves from one point to another in the "design space", the Toyota development approach uses a set of specifications to achieve a solution. Using set-based design, engineers and managers can test and evaluate numerous prototypes, which allows a more robust exploration of the design space and enables them to devise a combination of specifications to produce the most robust car or truck. Moreover, if problems are encountered during development, a set of possible alternatives exists (McElroy, 1994).

This research provides a number of interesting topics for further study. Could a set-based concurrent engineering approach be applicable in the aerospace industry? How would a set-based approach be implemented? What changes would be necessary to successfully implement a set-based development approach?

Robust Technology Development

Robust technology development during the research and development phase can simplify and accelerate the product development process. Developed by Dr. Genichi Taguchi, the originator of quality engineering, Technology Development reduces the risks involved in introducing new technologies into a product and can significantly reduce the amount of time devoted to "tweaking" in a product development process (Ealey, 1994; Clausing, 1994). Technology Development ensures that a technology is capable of overcoming downstream forces that can introduce variability in the end product (Ealey, 1994).

According to Dr. Taguchi, ensuring "origin quality" through Technology Development is the most powerful application of quality engineering, providing the greatest leverage for saving cost and time in the product development process (Ealey, 1994).[8] Since less funding is being allocated for developing and procuring new defense

systems, the quality of the output of research and development activities has become even more important. It is important for our technology development programs to engage in robust Technology Development so that designers can merely "tune" the technologies to yield the desired outcome when it is desired to incorporate the technologies in a new product. For instance, since JAST (Joint Advanced Strike Technology) seeks to develop "bins of technology", a robust Technology Development approach would certainly benefit future aircraft development programs seeking to employ JAST-developed technologies. Clearly, the potential benefits and barriers to the application of Taguchi's Technology Development[7] concepts during research and development activities merit further study.

[7] Taguchi's Technology Development has already been applied successfully for several years at Nissan's Reliability Engineering Center. An example of its application can be found in Ealey (1994).

APPENDIX

Cleanroom Engineering Supplement

This appendix contains more detailed discussions of three aspects of Cleanroom engineering. Box structure design is discussed first. The following section discusses functional verification. The last section provides a more detailed description of the statistical testing practices of Cleanroom engineering.

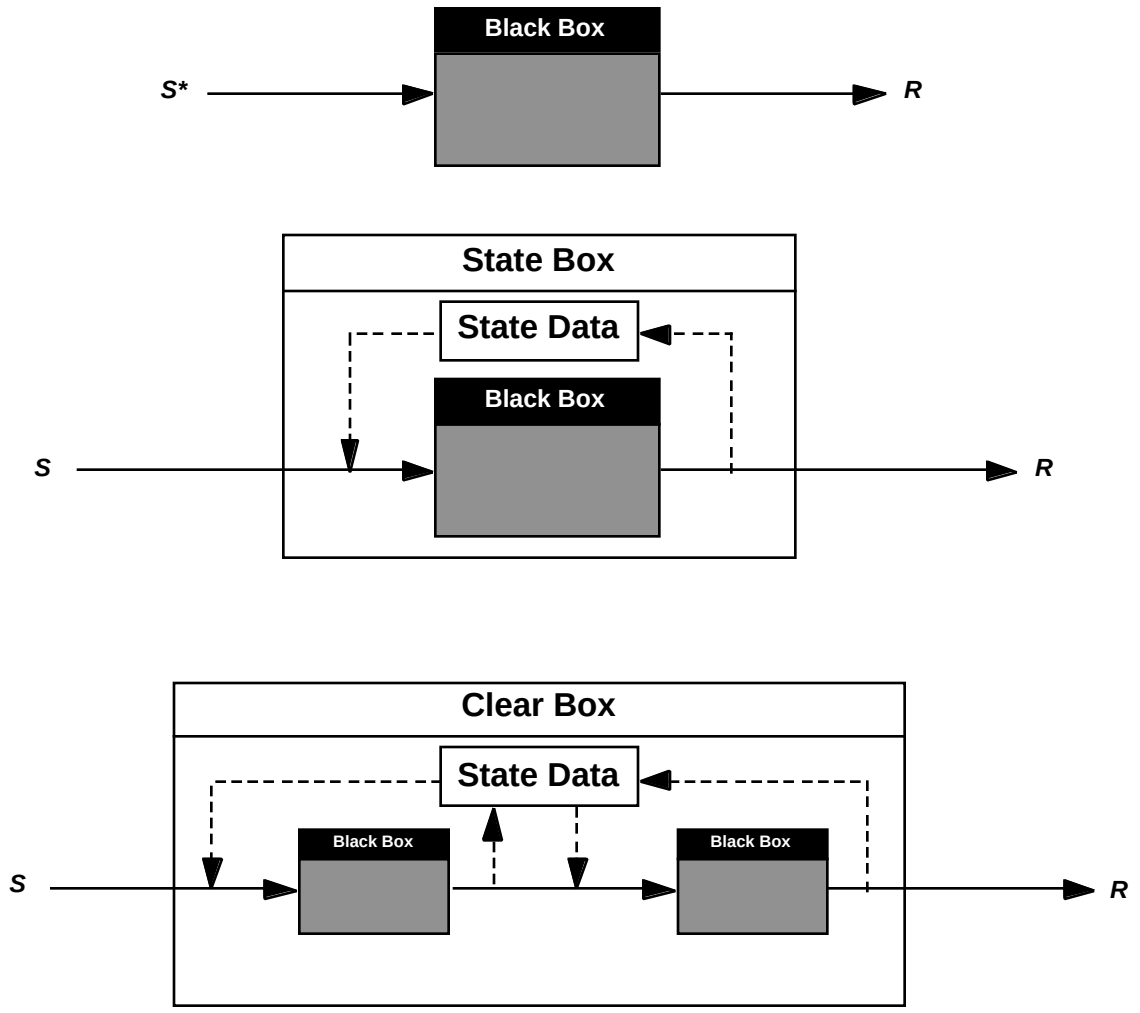
A.1 BOX STRUCTURE DESIGN

This section provides a more detailed discussion of box structure design methods. The different box structures—black boxes, state boxes, and clear boxes—are described first. The guiding principles of box structure design are discussed next. Finally, the box structure design algorithm is described. As a reminder, diagrams of the three box structures are shown again in Figure A.1.

A black box provides an implementation-free, object-oriented description of software. This box structure only describes the software's external behavior in terms of a mathematical function that maps a stimulus history, $\boldsymbol{S}^*$, to a response, $\boldsymbol{R}$. Since the black box view excludes all details of internal structures and operations, it also provides a description of the user's view of system behavior.

A state box provides a data-oriented view that begins to define implementation details by modifying the black box to represent responses in terms of the current stimulus,

Figure A.1: Box Structure Diagram



Note: The clear box structure is shown with a sequence process structure.

S, and state data that contains the stimulus histories. To form a state box, a black box is expanded by adding state data and state machine transitions to black box transitions. Thus, the state box contains a black box that accepts the external stimulus and the internal state data as its stimulus and produces both the external response and the new internal state which replaces the old state as its response. State box behavior can be described in the transition formula

(Stimulus, Old State) → (Response, New State)

A clear box provides a process-oriented view that completes the implementation details by modifying the state box view to represent responses in terms of the current stimulus, state data, and invocations of lower level black boxes. To form a clear box, a state box is expanded by adding procedure structures and delegating parts of the process to component black boxes. The processing can be defined in terms of three possible sequential structures—sequence, alternation, and iteration—and a concurrent structure.

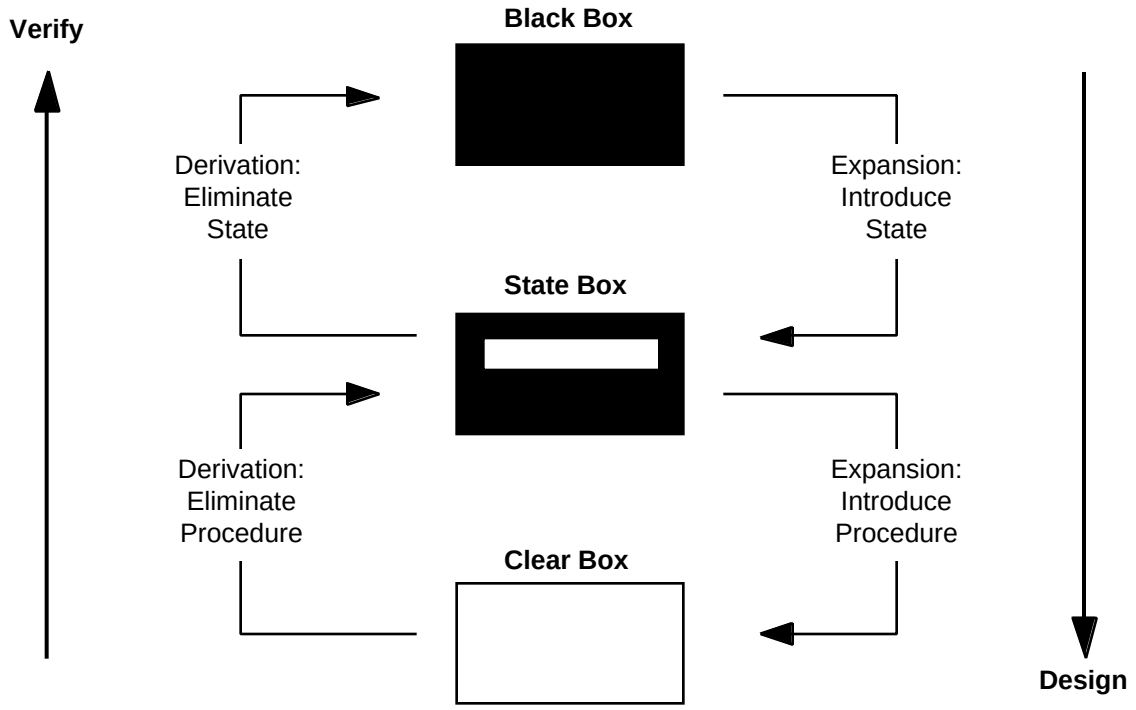
The relationships among the black box, state box, and clear box descriptions of a system or subsystem precisely define the tasks of *expansion* and *derivation*. Whereas it is an expansion task to design a state box from a black box or to design a clear box from a state box, it is a derivation task to abstract a black box from a state box or to abstract a state box from a clear box. An expansion does not produce a unique product since there are many state boxes that behave like a given black box and many clear boxes that behave like a given state box. A derivation, however, produces a unique product since there is only one black box that behaves like a given state box and only one state box that behaves like a given clear box (Mills et al., 1987). Expansion and derivation are the basis for box structure design and verification, as shown in Figure A.2.

The effective use of box structure design methods for the development of systems is guided by the application of six basic box structure principles: referential transparency, transaction closure, state migration, common services, correct design trail, and efficient verification.

Referential Transparency. This condition occurs when a black box is encapsulated by the clear box at the next higher level of the usage hierarchy. Each object is logically independent of the rest of the system and can be designed to satisfy a well defined "local" behavior specification. Referential transparency simplifies development and produces designs that are easy to enhance.

Transaction Closure. The transactions of a system or subsystem should be sufficient for acquiring and preserving all its state data, and its state data should be sufficient for completing all its transactions. The principle of transaction closure defines a systematic, iterative specification process to ensure that a sound and **complete** set of transactions is identified to achieve the required system behavior. The result is that the required stimuli,

Figure A.2: Box Structure Expansion and Derivation



data, and transactions are available at each stage of the design to generate the desired behavior.

State Migration. State data is identified and stored in the data abstraction at the lowest level in the box structure hierarchy that includes all references to that data. The result is that state data can easily be transferred to the lowest feasible level.

Common Services. A common service is a data abstraction which is described in a separate box structure hierarchy, and used in other box-structured systems. System parts with multiple uses are defined as common services for reusability. In the same way, predefined common services, such as database management systems and reuse objects, are incorporated into the design in a natural manner. The results are smaller systems and designs which accommodate reuse objects.

Correct Design Trail. It is important to insure consistency in the entire design trail when correcting an error. If the changes to a previous design document are trivial, the corrections

can be performed right away without stopping the current design process. However, if the necessary changes are major, the design process should be stopped until all the corrections to the previous design documents are made.

Efficient Verification. It is only necessary to verify what is changed from one refinement to the next since all elements of the design are referentially transparent. In addition, if the design refinements are conducted in small steps instead of large leaps, each refinement can be shown to be correct by direct assertion.[1] Otherwise, verification might require the use of rigorous proofs which are significantly more time intensive than the direct assertion approach.

The box structure design algorithm begins with defining and verifying the black box function for the system. This task is usually accomplished by the specification team who delivers it to the development team. Once the black box has been defined and verified, the design is refined by expanding the black box into a state box. Before the state box is expanded into a clear box, a black box is derived from the state box by eliminating references to state data in the state box functions and comparing it to the original black box. If the two black boxes are equivalent, then the state box design is verified as correct. Otherwise, the state box design must be corrected.

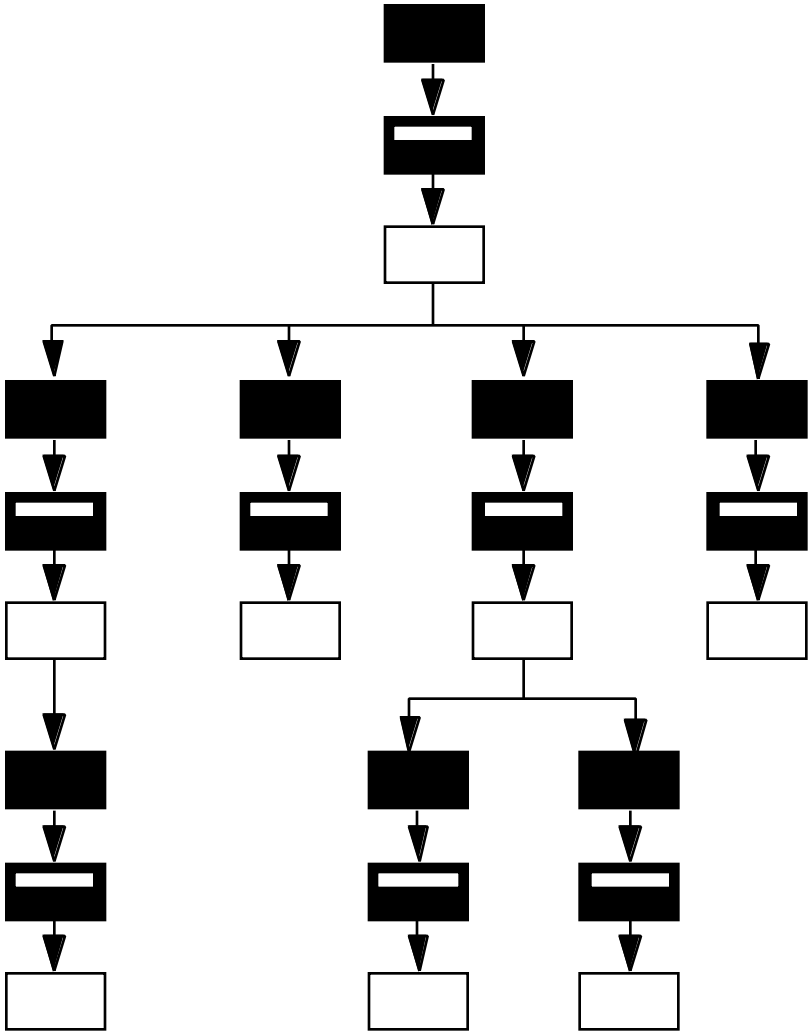
After the state box refinement has been verified, the design is refined once more by expanding the state box into a clear box. Again, the clear box design must be verified by deriving a state box from the clear box by eliminating references to procedure and the internal black boxes it encapsulates and comparing it with the original state box. If the state boxes are the same, the design process can continue. Otherwise, clear box design must be iterated until a design can be verified successfully.

Following verification of the refinement, this same process is repeated for each of the internal black boxes in a stepwise refinement process that ends when all the remaining internal black boxes represent either single commands of the destination language or subsystems that have been implemented in a previous increment. This top-down design process produces a box structure hierarchy, such as the one shown in Figure A.3.

---

[1] Refining designs in small steps typically creates simple software designs that are typically correct. Refining designs in large leaps can increase the chance that errors will be introduced into the design.

Figure A.3: Box Structure Hierarchy



After the top-down box structure design is completed, the clear boxes can be implemented. Depending on the nature of the system under development, the actual clear box implementation could be accomplished through an integration of hardware, software, and human behavior (SET, 1993). For software development, clear boxes are translated into code through stepwise refinement. The implementation must be verified as correct and consistent with the clear box design. The software code is verified by demonstrating the equivalence of the program and the design represented by the clear box refinement. While system design proceeds in a top-down fashion, the implementation of the design is accomplished in a bottom-up fashion. Designing top-down and then coding bottom-up allows the developers to exploit fully the principle of common services during the design phase and generalize the common services as much as possible during the coding phase.

A.2 FUNCTIONAL VERIFICATION

In Cleanroom engineering, functional verification is used instead of unit debugging. Once a clear box has been refined into code, the development team uses functional verification to help structure a proof that the refinement correctly implements the clear box design. If multiple steps were taken to refine the clear box design to code, the current refinement would be verified against the last refinement. The principle of referential transparency allows these smaller proofs to be easily accumulated into a proof for a large program. Unlike the unit debugging practices that are traditionally used, functional verification is scalable. Experience demonstrates that people are able to master these ideas with little difficulty and construct proofs for very large software systems (Cobb and Mills, 1990).

These functional verifications typically yield surprising improvements in design, even for the best software engineers. As a result, the developed software can be smaller and faster than previously thought possible, delivering more functionality per instruction. In addition, using functional verification allows quality to be designed into the software. According to Cobb and Mills (1990), functional verification leaves only two to five defects per thousand lines of code to be fixed in later phases of the life cycle whereas debugging leaves 10 to 30 defects per thousand lines of code. In contrast to functional verification, debugging attempts to test quality into the product. However, since more than 15 percent of the corrections merely introduce newer, deeper errors, testing quality into software is not possible (SET, 1993).[2]

Functional verification and testing seem to exhibit a high degree of synergy. Functional verification eliminates defects that tend to be difficult to detect with testing. The defects that remain after inspections and functional verification are generally the type that can be easily detected with testing. The combination of inspections, functional verification, and testing can eliminate more than 99 percent of all defects (Head, 1994). Table A.1 summarizes the defect removal performance for a range of different defect detection strategies.

---

[2] DeMarco (1982) contains an excellent analysis which demonstrates the validity of this point. Testing seems to be capable of eliminating half of the software defects. However, this factor of two improvement is overwhelmed by the extreme variability in the quality of software being produced today.

Table A.1: Defect Removal Percentages of Different Strategies

Detection Strategy	% Defects Removed
Testing	50%
Inspections	60%
Inspections + Testing	85%
Inspections + Functional Verification	90%
Inspections + Functional Verification + Testing	>99%

Source: Head (1994)

A.3 STATISTICAL TESTING

Cleanroom engineering makes use of statistical usage testing to certify the reliability of the developed software in terms of its MTTF (Mean Time To Failure). The application of rigorous statistical theory allows both quality control of the software being developed and process control over the development of the software.

The testing approach used in Cleanroom projects differs from the prevailing approach of coverage testing. The goal of Cleanroom testing is to maximize the expected MTTF of the software under development. Since coverage testing is just as likely to discover rare execution failures as it is to discover frequent execution failures, an alternative strategy that focuses on detecting frequent failures is needed. This need is satisfied by usage testing since its utilization of a statistical usage profile allows the formulation of tests that are representative of expected usage. In fact, for the purposes of increasing the MTTF of software, usage testing has been determined to be 21 times more effective than coverage testing (Cobb and Mills, 1990). Usage testing also typically takes less time than coverage testing.

The usage profile is represented by a Markov model specifying the probability of moving from each usage state to all other usage states. The model describes every possible state that a system can be in, identifies all the different actions that the user could take in each state, and assigns probabilities to each possible action in each state. The usage profile can be expressed in a state transition diagram or a matrix.

The requirement specifications and usage profile, which are developed by the specification team, are used by the certification team to develop a set of random test cases which are representative of the expected usage. After the current increment is coded and delivered to the certification team, the current increment is integrated with previous increments, if any, and the test scenarios are executed. The error history is evaluated with a mathematical model designed to predict how many more defects the user might encounter in a certain period of time with a certain number of uses (Head, 1994). As soon as the reliability model indicates that the developed software meets or exceeds the desired quality level with a sufficiently small degree of uncertainty, testing is considered complete, and the product can be safely released.[3]

Software must be minimally reliable for statistical testing to be valid. Applying statistical testing to software developed with typical defect densities would cause the statistical reliability models to blow up. If the model does not blow up, its predictions are usually extremely unfavorable (Head, 1994). According to Cobb and Mills (1990), defect densities of five defects per thousand lines of code or less can be tolerated without invalidating the application of statistical MTTF estimation.[4]

---

[3] Currit et al. (1986) describes several statistical models for certifying the reliability of software.

[4] The reader may recall that this figure of 5 defects/KLOC was mentioned earlier during a description of the Cleanroom principle of defect prevention. Apparently, discarding software with high defect densities is done not only to motivate designers to develop defect-free software but for statistical reasons as well.

BIBLIOGRAPHY

Adams, Edward N. (1984). Optimizing Preventive Service of Software Products. *IBM Journal of Research and Development*, January 1984.

Alberts, D. S. (1976). The Economics of Quality Assurance. *National Computer Conference*.

Aldinger, Charles (1994). Perry Promises Arms Acquisition Reform. Article posted in clari.news.usa.military. Reuters, May 12, 1994.

Amer, Kenneth B., Raymond W. Prouty, Greg Korkosz, and Doug Fouse (1992). *Lessons Learned During the Development of the AH-64A Apache Attack Helicopter*. Santa Monica: RAND. RP-105.

Anderson, Christine and Merlin Dorfman (1991). Preface. In Anderson, Christine and Merlin Dorfman, editors, *Aerospace Software Engineering*. Volume 136. Progress in Aeronautics and Astronautics. Washington, DC: AIAA.

Anderson, Michael G. (1992). *The Air Force Rapid Response Process: Streamlined Acquisition During Operations Desert Shield and Desert Storm*. Santa Monica: RAND. N-3610/3-AF.

Anson, Sir Peter and Dennis Cummings (1992). The First Space War: The Contribution of Satellites to the Gulf War. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Argyris, Chris (1991). Teaching Smart People How to Learn. *Harvard Business Review*. May-June.

Argyris, Chris (1993). Education for Leading-Learning. *Organizational Dynamics*. Winter.

Arquilla, John and David Ronfeldt (1992). *Cyberwar Is Coming!* Santa Monica: RAND. P-7791.

Bankes, Steve and Carl Builder (1992). Seizing the Moment: Harnessing the Information Technologies. *The Information Society,* Vol. 8, 1992.

Bodilly, Susan J. (1993a). *Case Study of Risk Management in the USAF B-1B Bomber Program.* Santa Monica: RAND. N-3616-AF.

Bodilly, Susan J. (1993b). *Case Study of Risk Management in the USAF LANTIRN Program.* Santa Monica: RAND. N-3617-AF.

Boehm, Barry W. (1976). Software Engineering. *IEEE Transactions on Computers.* C-25, December 12, 1976.

Boehm, Barry W. (1981). *Software Engineering Economics*. Englewood Cliffs: Prentice-Hall, Inc.

Boehm, Corrado and Giuseppe Jacopini (1966). Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. *Comm. ACM,* Vol. 9, No. 5, May 1966.

Bond, David F. (1991). Cost, Supportability Key to Boeing Sikorsky LH Award. *Aviation Week & Space Technology,* April 15, 1991.

Bowen, H. Kent, Kim B. Clark, Charles A. Holloway, and Steven C. Wheelwright (1994). Development Projects: The Engine of Renewal. *Harvard Business Review,* September-October.

Bowie, Christopher, Fred Frostic, Kevin Lewis, John Lund, David Ochmanek, and Philip Propper (1993). *The New Calculus: Analyzing Airpower's Changing Role in Joint Theater Campaigns.* Santa Monica: RAND. MR-149-AF.

Buck, Joseph, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt (1994). Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. International Journal of Computer Simulation, special issue on Simulation Software Development, January 1994.

Camm, Frank (1993a). *The Development of the F100-PW-220 and F110-GE-100 Engines: A Case Study of Risk Assessment and Risk Management.* Santa Monica: RAND. N-3618-AF.

Camm, Frank (1993b). *The F-16 Multinational Staged Improvement Program: A Case Study of Risk Assessment and Risk Management.* Santa Monica: RAND. N-3619-AF.

Campen, Alan D. (1992a). Information Systems and Air Warfare. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Campen, Alan D. (1992b). Iraqi Command and Control: The Information Differential. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Campen, Alan D. (1992c). Electronic Templates. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Campen, Alan D. (1992d). Communications Support to Intelligence. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Campen, Alan D. (1992e). Introduction. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Cava, Jeffrey (1993). I-War. In Cava, Jeffrey, writer, *Soft Kill*. CD-ROM. Xiphias.

Clark, Kim B. and Takahiro Fujimoto (1991). *Product Development Performance: Strategy, Organization, and Management in the World Auto Industry*. Boston: Harvard Business School Press.

Clausing, Don (1994). *Total Quality Development: A Step-by-Step Guide to World-Class Concurrent Engineering*. New York: ASME Press.

Cobb, Richard H. and Harlan D. Mills (1990). Engineering Software under Statistical Quality Control. *IEEE Software*, November 1990.

Currit, P. Allen, Michael Dyer and Harlan D. Mills (1986). Certifying the Reliability of Software. *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, January 1986.

Cusumano, Michael A. (1991). *Japan's Software Factories: A Challenge to U.S. Management*. New York: Oxford University Press.

DeMarco, T. (1982). *Controlling Software Projects*. New York: Yourdon Press.

Department of Commerce (1994). *U.S. Industrial Outlook 1994*. Washington, DC: U.S. Government Printing Office.

Department of Defense (1992). *The Conduct of the Persian Gulf War*. Washington, DC: U.S. Government Printing Office.

Dijkstra, Edsger W. (1969). Structured Programming. In Buxton, J. N. and B. Randell, editors, *Software Engineering Techniques*, NATO Science Committee, Rome.

Ealey, Lance (1994). Robust Technology. *Automotive Industries*, Vol. 174, No. 8, August.

Electronic Industries Association (1991). *Balancing National Security With Realities of the 1990s: Ten-Year Forecast of Defense Electronic Opportunities (FYs 1992-2001)*. Washington, DC.

Electronic Industries Association (1988). *DoD Computing Activities and Programs: Ten-Year Market Forecast Issues, 1985-1995*. Washington, DC.

Endres, A. B. (1975). An Analysis of Errors and Their Causes in System Programs. *IEEE Transactions on Software Engineering*, June 1975.

Fagan, M. E. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, Vol. 15, No. 3.

Fulghum, David A. (1992). "Secret Carbon-Fiber Warheads Blinded Iraqi Air Defenses." Aviation Week & Space Technology, April 27, 1992.

Fulghum, David A. (1993a). Major Changes Planned for Wild Weasel Force. *Aviation Week & Space Technology*, July 5, 1993.

Fulghum, David A. (1993b). USAF Holds Pre-JDAM Test. *Aviation Week & Space Technology*, July 5, 1993.

Fulghum, David A. (1993c). Loh Outlines Bomber Plans. *Aviation Week & Space Technology*, July 5, 1993.

Fulghum, David A. (1993d). Talon Lance Gives Aircrews Timely Intelligence from Space. Aviation Week & Space Technology, August 23, 1993.

Fulghum, David A. (1993e). USAF Wing Takes Innovations Overseas. *Aviation Week & Space Technology*, December 13/20, 1993.

Fulghum, David A. (1994a). New Air Defenses Worry SEAD Experts. *Aviation Week & Space Technology*, January 17, 1994.

Fulghum, David A. (1994b). Specialists Debate Merits of Wild Weasel Replacements. Aviation Week & Space Technology, January 17, 1994.

Fulghum, David A. (1994c). JAST Plans Envisions ASTOVL Prototype. *Aviation Week & Space Technology*, February 28, 1994.

Fulghum, David A. (1994d). New Wartime Roles Foreseen for JSOW. *Aviation Week & Space Technology*, February 28, 1994.

Fulghum, David A. (1994e). Stealthy TSSAM Aces Tests But Faces Budget Battle. Aviation Week & Space Technology, September 12, 1994.

Fulghum, David A. and John D. Morrocco (1994). Deutsch Demands Cuts, Services Scramble Anew. *Aviation Week & Space Technology*, August 29, 1994.

Gebman, J. R., D. W. McIver, and H. L. Shulman (1989). *A New View of Weapon System Reliability and Maintainability*. Santa Monica: RAND. R-3604/2-AF.

Gelernter, David (1991). *Mirror Worlds, or the Day Software Puts the Universe in a Shoebox...How It Will Happen and What It Will Mean.* New York: Oxford University Press.

General Accounting Office (1992). *Embedded Computer Systems: Significant Software Problems on C-17 Must Be Addressed*. Washington, DC: U.S. Government Printing Office. GAO/IMTEC-92-48. May 1992.

Glennan, Thomas K., Susan J. Bodilly, Frank Camm, Kenneth R. Mayer, and Timothy J. Webb (1993). *Barriers to Managing Risk in Large Scale Weapon System Development Programs*. Santa Monica: RAND. MR-248-AF.

Hammer, Michael and James Champy (1993). *Reengineering the Corporation: A Manifesto for Business Revolution*. New York: HarperBusiness.

Head, Grant (1994). Six-Sigma Software Using Cleanroom Software Engineering Techniques. *Hewlett-Packard Journal*, June 1994.

Hou, Alexander C. (1994). *Meeting U.S. Defense Needs in the Information Age: An Evaluation of Selected Complex Electronic System Development Methodologies*. Master's Thesis at the Massachusetts Institute of Technology. Submitted on November 30, 1994 for February 1995 degree list.

Hughes, David (1994a). AWACS Data Fusion Under Evaluation. *Aviation Week & Space Technology*, March 7, 1994.

Hughes, David (1994b). Mitre, Air Force Explore Data Fusion for Joint-STARS. *Aviation Week & Space Technology*, March 7, 1994.

Hyde, John Paul, Johann W. Pfeiffer, and Toby C. Logan (1992). CAFMS Goes to War. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

IBM (1990). Software-First Life Cycle: Final Definition. *STARS CDRL No. 01240*. January 5, 1990.

Jones, T.C. (1978). Measuring Programming Quality and Productivity. *IBM Systems Journal*, Vol. 17, No. 1.

Jones, T. C. (1981). Defect Removal: A Look at the State of the Art. *ITT ComNet*, Vol. 1, No. 3. December 1981.

Kalavade, Asawaree (1991). Hardware/Software Codesign Using Ptolemy: A Case Study. UC Berkeley Master's Thesis.

Kalavade, Asawaree and Edward A. Lee (1992). Hardware/Software Co-Design Using Ptolemy—A Case Study. *Proceedings of the First International Workshop on Hardware/Software Codesign*. September 1992.

Kalavade, Asawaree and Edward A. Lee (1993). A Hardware-Software Codesign Methodology for DSP Applications. *IEEE Design & Test of Computers*. September 1993.

Kandebo, Stanley W. (1991). Boeing Sikorsky LH Technologies Will Increase Army Combat Capability. *Aviation Week & Space Technology*, April 15, 1991.

Kandebo, Stanley W. (1994). Cruise Missile Updates Pending. *Aviation Week & Space Technology*, January 17, 1994.

Krepinevich, Andrew F. (1994). Keeping Pace with the Military-Technological Revolution. *Issues in Science and Technology*, Summer 1994.

Larson, Eric V. (1990). Technological Risk: The Case of the Tomahawk Cruise Missile. Santa Monica: RAND. P-7672-RGS.

Lawrence, Paul R. and Jay W. Lorsch (1967). *Differentiation and Integration in Complex Organizations*. Boston: Graduate School of Business Administration, Harvard University.

Lawrence, Paul R. and Jay W. Lorsch (1986). *Organization and Environment: Managing Differentiation and Integration*. Boston: Harvard Business School Press.

Lawrence, Paul R., Louis B. Barnes, and Jay William Lorsch (1976). *Organizational Behavior and Administration: Cases and Readings*. Homewood, IL: R. D. Irwin.

Lenorovitz, Jeffry M. (1991a). AWACS Played Critical Role in Allied Victory Over Iraq. Aviation Week & Space Technology, March 4, 1991.

Lenorovitz, Jeffry M. (1991b). F-16As Prove Usefulness in Attack Role Against Iraqi Targets in Desert Storm. *Aviation Week & Space Technology*, April 22, 1991.

Leonard-Barton, Dorothy, H. Kent Bowen, Kim B. Clark, Charles A. Holloway, and Steven C. Wheelwright (1994). How to Integrate Work and Deepen Expertise. Harvard Business Review, September-October.

Ling, James G. (1993). Principles of Lean Manufacturing. Internal Lean Aircraft Initiative Memorandum. October 1993.

Lorell, Mark A. (1989). *The Use of Prototypes in Selected Foreign Fighter Aircraft Development Programs: Rafale, EAP, Lavi, and Gripen*. Santa Monica: RAND. R-3687-P&L.

Malone, Thomas W. and John F. Rockart (1991). Computers, Networks and the Corporation. *Scientific American*, September 1991.

Maras, M. et al. (1994). Rapid Protoyping and Integrated Design System for Software Development of GN&C Systems. 17th Annual AAS Guidance and Control Conference. February 2-6, 1994.

Martin, J. (1982). *Application Development Without Programmers.* Englewood Cliffs: Prentice-Hall, Inc.

Mayer, Kenneth (1993). *The Development of the Advanced Medium Range Air-to-Air Missile: A Case Study of Risk and Reward in Weapon System Acquisition.* Santa Monica: RAND. N-3620-AF.

McElroy, John (1994). Toyota's Product Development Paradox. *Automotive Industries*, Vol. 174, No. 8, August.

Mills, Harlan D. (1986). Structured Programming: Retrospect and Prospect. *IEEE Software*, November 1986.

Mills, H. D., Linger, R. C., and Hevner, A. R. (1987). Box Structured Information Systems. *IBM Systems Journal*, Vol. 26, No. 4.

Mills, Harlan D. and Jesse H. Poore (1988). Bringing Software Under Statistical Quality Control. *Quality Progress*, Vol. 21, No. 11, November 1988.

Mills, Harlan D. (1991). Cleanroom: An Alternative Software Development Process. In Anderson, Christine and Merlin Dorfman, editors, *Aerospace Software Engineering*. Volume 136. Progress in Aeronautics and Astronautics. Washington, DC: AIAA.

Momyer, General William W. (1978). *Air Power in Three Wars*. Washington, DC: U.S. Government Printing Office.

Morrocco, John D. (1993). Pentagon Pushes TSSAM Despite Technical Problems. Aviation Week & Space Technology, October 18, 1993.

Morrocco, John D. (1994). U.S. Trains for Peacekeeping. *Aviation Week & Space Technology*, April 25, 1994.

Morrocco, John D. and David A. Fulghum (1994). Radar-Killing F-15 May Fall to Budget Ax. *Aviation Week & Space Technology*, September 12, 1994.

Nordwall, Bruce D. (1993). Companies Reduce Solder to Increase Reliability. *Aviation Week & Space Technology*, December 6, 1993.

Myers, G. J. (1976). *Software Reliability: Principles and Practices.* New York: John Wiley & Sons, Inc.

Pagonis, Lt. General William G. (1992). *Moving Mountains.* Written with Jeffrey L. Cruikshank. Boston: Harvard Business School Press.

Paulk, Mark C., Bill Curtis, Mary Beth Chrissis, and Charles V. Weber (1993a). *Capability Maturity Model for Software, Version 1.1*. CMU/SEI-93-TR-24. February 1993.

Paulk, Mark C., Charles V. Weber, Suzanne M. Garcia, Mary Beth Chrissis, and Marilyn Bush (1993b). *Key Practices of the Capability Maturity Model, Version 1.1*. CMU/SEI-93-TR-25. February 1993.

Rich, Ben R. and Leo Janos (1994). *Skunk Works*. Boston: Little, Brown & Company.

Rich, Michael and Edmund Dews (1986). *Improving the Military Acquisition Process: Lessons from Rand Research.* Santa Monica: RAND. R-3373-AF/RC.

Richards, Mark A. (1994). The Rapid Prototyping of Application Specific Signal Processors (RASSP) Program: Overview and Accomplishments. July 1994.

Ronfeldt, David (1991). *Cyberocracy, Cyberspace, and Cyberology: Political Effects of the Information Revolution*. Santa Monica: RAND. P-7745.

SAF/AQK (1992). Guidelines for the Successful Acquisition of Computer Dominated Systems and Major Software Developments. February 20, 1992.

Schwarzkopf, General H. Norman (1992). *It Doesn't Take a Hero*. Written with Peter Petre. New York: Bantam Books.

Scott, William B. (1994). Military Space "Reengineers". *Aviation Week & Space Technology*, August 15, 1994.

Senge, Peter M. (1990). *The Fifth Discipline: The Art and Practice of the Learning Organization.* New York: Doubleday.

Sherer, S. Wayne, Paul G. Arnold, and Ara Kouchakdjian (1994). Successful Process Improvement Effort Using Cleanroom Software Engineering.

Shooman, M. L. (1983). *Software Engineering—Design Reliability and Management.* New York: McGraw-Hill, Inc.

Software Engineering Technology, Inc. (1993). Cleanroom System and Software Engineering: A Technical and Management Abstract. April 1993.

Sproull, Lee and Sara Keisler (1991). *Connections: New Ways of Working in the Networked Organization*. Cambridge: MIT Press.

Steele, Robert D. (1993). The Transformation of War and the Future of the Corps. In Cava, Jeffrey, writer, *Soft Kill*. CD-ROM. Xiphias.

Suh, Nam P. (1990). *The Principles of Design*. New York: Oxford University Press.

Sun Tzu (1971). *The Art of War*. Translated by Samuel B. Griffith. New York: Oxford University Press.

Swalm, Thomas S. (1992). Joint STARS in Desert Storm. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Thayer, T. A. et al. (1978). *Software Reliability: A Study of Large Project Reality*. New York: North Holland.

Toffler, Alvin (1990). *Powershift: Knowledge, Wealth, and Violence at the Edge of the 21st Century.* New York: Bantam Books.

Toffler, Alvin and Heidi Toffler (1993). *War and Anti-War*. Boston: Little, Brown and Company.

Toma, Joseph S. (1992). Desert Storm Communications. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Uhde-Lacovara, Jo, Daniel Weed, Bret McCleary, and Ron Wood (1994). The Rapid Development Process Applied to Soyuz Simulation Production.

Ward, Allen, Jeffery K. Liker, John J. Cristiano, and Durward K. Sobek, II (1994). The Second Toyota Paradox: How Delaying Decisions Can Make Better Cars Faster. August 26, 1994.

Wentz, Larry K. (1992). Communications Support for the High Technology Battlefield. In Campen, Alan D., editor, *The First Information War*. Fairfax: AFCEA.

Womack, James P., Daniel T. Jones, and Daniel Roos (1991). *The Machine That Changed the World.* New York: HarperPerennial.