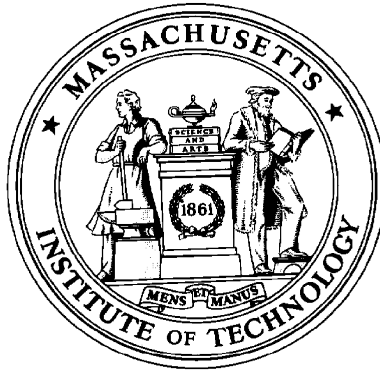


Report – LEAN 96-02

The Software Factory:
Integrating CASE Technologies to Improve Productivity

Prepared by: **José Menendez**

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

July 1996

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 authors take full responsibility.

© Massachusetts Institute of Technology, 1996

TABLE OF CONTENTS

THE SOFTWARE FACTORY............................ERROR! BOOKMARK NOT DEFINED.

CHAPTER 1: INTRODUCTION 1

CHAPTER 2: CASE TECHNOLOGY AND THE SOFTWARE FACTORY 5

Software Development in the Aerospace Industry....................................7

The Software Factory....................................11

Integrating CASE Technology into a Factory Support Environment....................................13

CASE Tool Integration....................................15

Automatic Code Generation....................................18

Strategic Management....................................21

CHAPTER 3: CASE TECHNOLOGY DEVELOPMENT IN THE AEROSPACE INDUSTRY....................................25

United Technologies / Pictures-to-Code....................................26

The Pictures-to-Code Process....................................26

The Pictures-to-Code Toolset....................................28

Automated Code Generation with PtC....................................29

General Electric / BEACON....................................34

The BEACON Integrated Environment....................................34

Integrated Systems, Inc. / MATRIXx....................................36

The MATRIXx Product Family....................................36

McDonnell Douglas / RAPIDS....................................37

The RAPIDS Software Development Process....................................37

The RAPIDS Integrated Environment....................................39

NASA-JSC / Rapid Development Lab....................................41

Lockheed / LEAP....................................42

The LEAP Integrated Environment....................................42

Honeywell and the Domain Specific Software Architecture Program....................................44

The DSSA Toolset....................................45

Charles Stark Draper Laboratory / CSDL CASE System....................................46

The CSDL CASE System....................................46

Verilog / SAO+SAGA....................................48

The SAO+SAGA Integrated Environment....................................48

Summary....................................49

CHAPTER 4: INDUSTRY EXPERIENCES WITH SOFTWARE FACTORY DEVELOPMENT 51

United Technologies / Pictures-to-Code....................................51

Groundrules and Methodology for Pictures-to-Code Evaluation....................................51

The Pictures-to-Code Process....................................52

Traditional versus Pictures-to-Code....55
Continuous Process and Toolset Improvement....60
The Japanese Software Factories....63
Improving the Productivity and Quality of Software Development....66
The Benefits and Implementation of the Software Factory....70
CHAPTER 5: CONCLUSIONS....75
Observations and Findings....75
Areas for Further Study....78
CHAPTER 6: BIBLIOGRAPHY....79

LIST OF TABLES

TABLE 2.1 MIL-STD-498 SOFTWARE DEVELOPMENT ACTIVITIES....10
TABLE 3.1 ENVIRONMENTS AND CASE TECHNOLOGY IN THE AEROSPACE INDUSTRY....25
TABLE 3.2 COMMON CHARACTERISTICS....50
TABLE 4.1 SOFTWARE PRODUCTIVITY....64
TABLE 4.2 SOFTWARE QUALITY....65
TABLE 4.3 % EFFORT BY SOFTWARE DEVELOPMENT ACTIVITY....65
TABLE 4.4 ENCODING ERRORS IN AIRBUS SOFTWARE....68
TABLE 4.5 SHARED BENEFITS OF SOFTWARE FACTORY DEVELOPMENT....70
TABLE 4.6 SOFTWARE FACTORY IMPLEMENTATION....73
TABLE 5.1 LEAN PRINCIPLES ENABLED BY THE SOFTWARE FACTORY....75
TABLE 5.2 OBSERVATIONS AND FINDINGS ON SW FACTORY DEVELOPMENT AND TECHNOLOGIES....76
TABLE 5.3 DISTRIBUTION OF F-22 SOFTWARE DOMAIN....78

LIST OF FIGURES

FIGURE 1.1 GROWTH OF AIRCRAFT SOFTWARE 1
FIGURE 1.2 THE THREE LEVERAGE POINTS OF SOFTWARE DEVELOPMENT 2
FIGURE 2.1 SOFTWARE PROBLEM TYPES DURING WEAPON SYSTEM DEVELOPMENT....6
FIGURE 2.2 SOFTWARE DEVELOPMENT PROCESSES....6
FIGURE 2.3 DOD-STD-2167A SYSTEM DEVELOPMENT LIFE CYCLE....8
FIGURE 2.4 FORMS OF CASE INTEGRATION....14
FIGURE 2.5 LEVELS OF CASE TOOL INTEGRATION....16
FIGURE 2.6 NIST/ECMA REFERENCE MODEL....17
FIGURE 2.7 EVOLUTION OF CASE TECHNOLOGY....19
FIGURE 2.8 AUTOMATIC CODE GENERATION BRIDGES THE GAP IN CASE TECHNOLOGY 20
FIGURE 3.1 OVERVIEW OF THE COMMON DEVELOPMENT PROCESS FOR EEC SOFTWARE....27
FIGURE 3.2 FUNCTIONAL VIEW OF THE PICTURES-TO-CODE TOOLSET....30
FIGURE 3.3 CONTROL FLOW DIAGRAM AND STANDARD PART 31
FIGURE 3.4 DATA FLOW DIAGRAM AND STANDARD PART....32
FIGURE 3.5 THE BEACON ENVIRONMENT 35
FIGURE 3.6 SAMPLE SYMBOLS FROM BEACON....35
FIGURE 3.7 THE RAPIDS SOFTWARE DEVELOPMENT PROCESS....38
FIGURE 3.8 THE RAPIDS ENVIRONMENT....40
FIGURE 3.9 LOCKHEED ENVIRONMENT FOR AUTOMATIC PROGRAMMING....43
FIGURE 3.10 CSDL CASE SYSTEM 48
FIGURE 4.1 OVERVIEW OF THE COMMON DEVELOPMENT PROCESS FOR EEC SOFTWARE....54
FIGURE 4.2 % MODULE CYCLE TIME, 4000 CURRENT VS. 4000 GROWTH 58
FIGURE 4.3 DETECTED PROCESS ERRORS, 4000 CURRENT VS. 4000 GROWTH 58
FIGURE 4.4 MODULES CHANGED BY DEVELOPMENT METHOD VS. YEAR 59
FIGURE 4.5 % MODULE CYCLE TIME VS. YEAR, ALL LARGE COMMERCIAL PROGRAMS 59
FIGURE 4.6 DETECTED PROCESS ERRORS VS. YEAR, ALL LARGE COMMERCIAL PROGRAMS 60
FIGURE 4.7 % ERRORS BY PROGRAM VS. VERIFICATION ACTIVITY 62
FIGURE 4.8 % ERRORS BY VERIFICATION ACTIVITY FOR PTC METHOD 62
FIGURE 4.9 % ERRORS BY TYPE FOR PTC METHOD....63
FIGURE 4.10 COMPARISON OF PRODUCTIVITY INCREASES....67
FIGURE 4.11 COMPARISON OF REDUCTIONS TO ENCODING ERRORS....69

Chapter 1: Introduction

This report addresses the use of computer-aided software engineering (CASE) technology for the development of aircraft software. Real-time embedded software is becoming the key to implementing avionic systems functionality in all types of aircraft. Avionic systems in modern defense aircraft are highly complex. They are composed of multiple subsystems (navigation, radar, flight control, engine control, warfare systems, etc.) distributed over multiple processors throughout the aircraft. Embedded software, by implementing functionality within each subsystem and providing for overall integration, is both mission and safety critical. Embedded software is also growing exponentially in magnitude as shown in Figure 1.1 (Babel). This figure plots the on-board memory size for a number of defense aircraft.

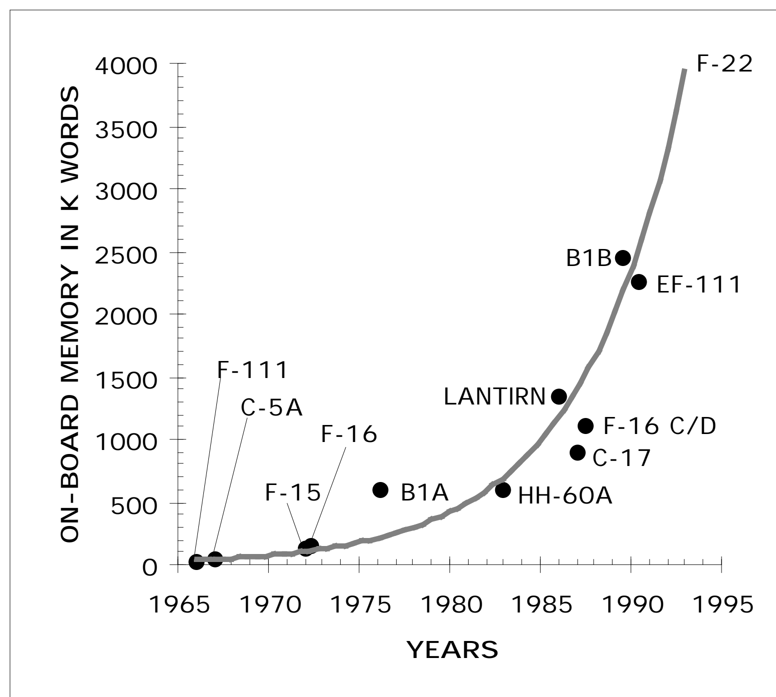


Figure 1.1 Growth of Aircraft Software

As the magnitude of software to be developed increases, software development continues to be plagued by problems that result in schedule overruns, cost overruns, poor quality software and software that fails to meet operational needs. These problems include incomplete requirements definition, changing requirements, lack of a clearly defined development method, improper design, inadequate testing and inadequate tools. The major challenge is to develop quality software in a reliable and repeatable manner while improving productivity.

People, process, and technology are three leverage points to meet this major challenge, Figure 1.2 (Over, 1992). Together these three leverage points are the major determinants of software quality and productivity. The focus of this report is technology, specifically integrating CASE technologies to create a software factory and automate software development.

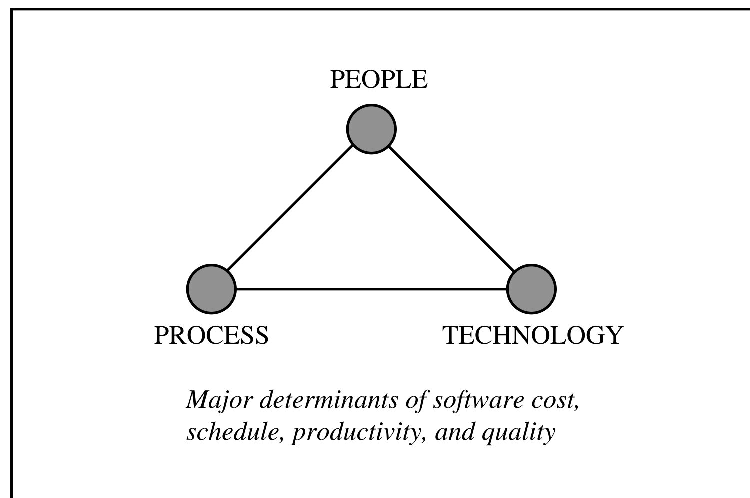


Figure 1.2 The Three Leverage Points of Software Development

Jones and Turkovich provides further emphasis on the need to focus on technology:

> In order to achieve significant reductions in software costs it is necessary to treat the software problem not merely as a managerial problem but as a technological problem. By doing so, the software development process becomes automation based, as opposed to automation assisted. This automation based system is supported with knowledge acquired by experts

> as they interact with the system in operation. In this way, software development leverages accumulated knowledge that can be used from application to application. Instead of people acting as bottlenecks in a flow to analyze functionality, they serve to fine-tune accumulated knowledge as appropriate on new projects.[1]

The remainder of this report follows this outline:

Chapter 2 introduces the concept of a software factory and discusses how CASE technology can be integrated into a software factory environment to improve quality and productivity.

Chapter 3 presents an overview of CASE technology development efforts in the aerospace industry and describes these technologies, which enable software factory development.

Chapter 4 discusses experiences in the aerospace industry with software factory development. A case study of United Technologies forms the bulk of this chapter. Productivity gains and additional benefits are discussed.

Chapter 5 concludes the report with a summary of observations and findings. Additional areas of study are recommended.

---

[1] Jones, Denise; and Turkovich, John; et al: "Automated Real-Time Software Development," Proceedings of the 3rd National Technology Transfer Conference & Exposition, NASA Conference Publication 3189, Volume 2, 1993, p. 184.

Chapter 2: CASE Technology and the Software Factory

To improve the quality and productivity of a software development process the major sources of errors and impediments to productivity must be addressed. The most basic phases of any software development process are requirements analysis, design, implementation, test and maintenance. The majority of software errors, as many as two-thirds, are introduced during requirements analysis and design, Figure 2.1 (Babel). The major impediment to productivity is the traditional hand-crafted approach to software development with specialists responsible for the different phases, making labor a significant cost element. The craft approach is characterized by diseconomies of scale, it is common for average productivity to decrease as the number of developers on a project increases. The labor intensive nature of software development also affects quality, which is difficult to maintain and control when large development teams are required.

Computer-Aided Software Engineering (CASE) uses tools to automate much of the software development process. Software process automation can reduce the labor requirement, significantly reduce errors introduced during implementation, and provide leverage toward the front-end stages of the process. Additionally, incorporation of CASE technologies can de-emphasize the coding and debugging tasks of implementation, and thus shift focus to requirements analysis and design. In addition, providing automated support for analysis and design can eliminate many errors that occur during the requirements stage of development. The goal of implementing CASE technology is to achieve an ideal software development process wherein requirements and design specification are directly translated into error-free software that does not require testing or maintenance, Figure 2.2 (Fischer, 1988).

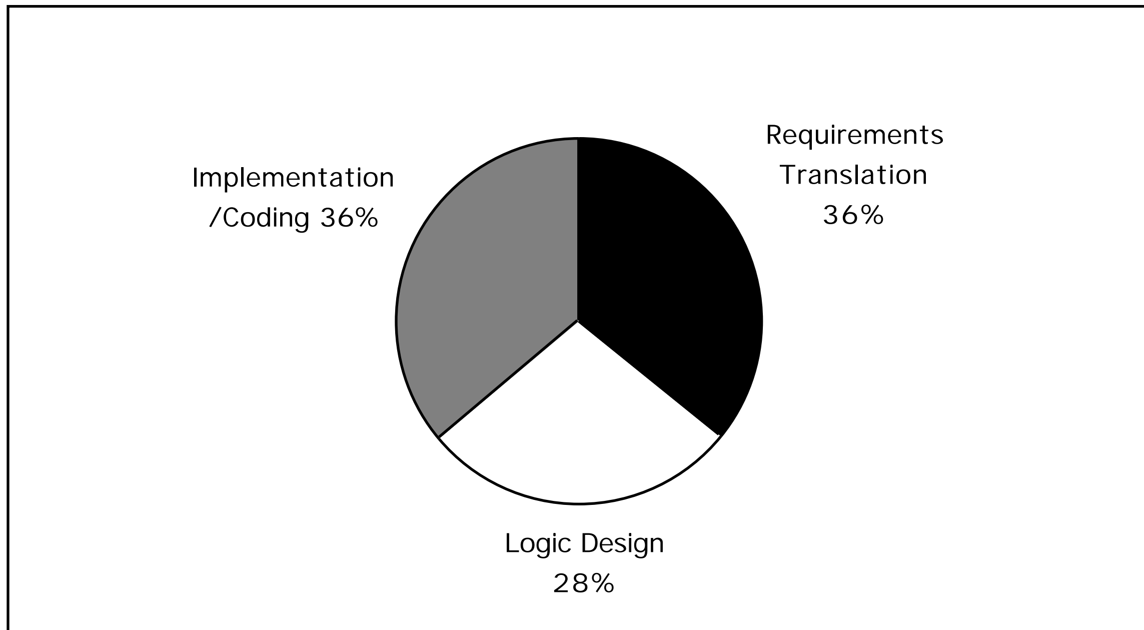


Figure 2.1 Software Problem Types During Weapon System Development

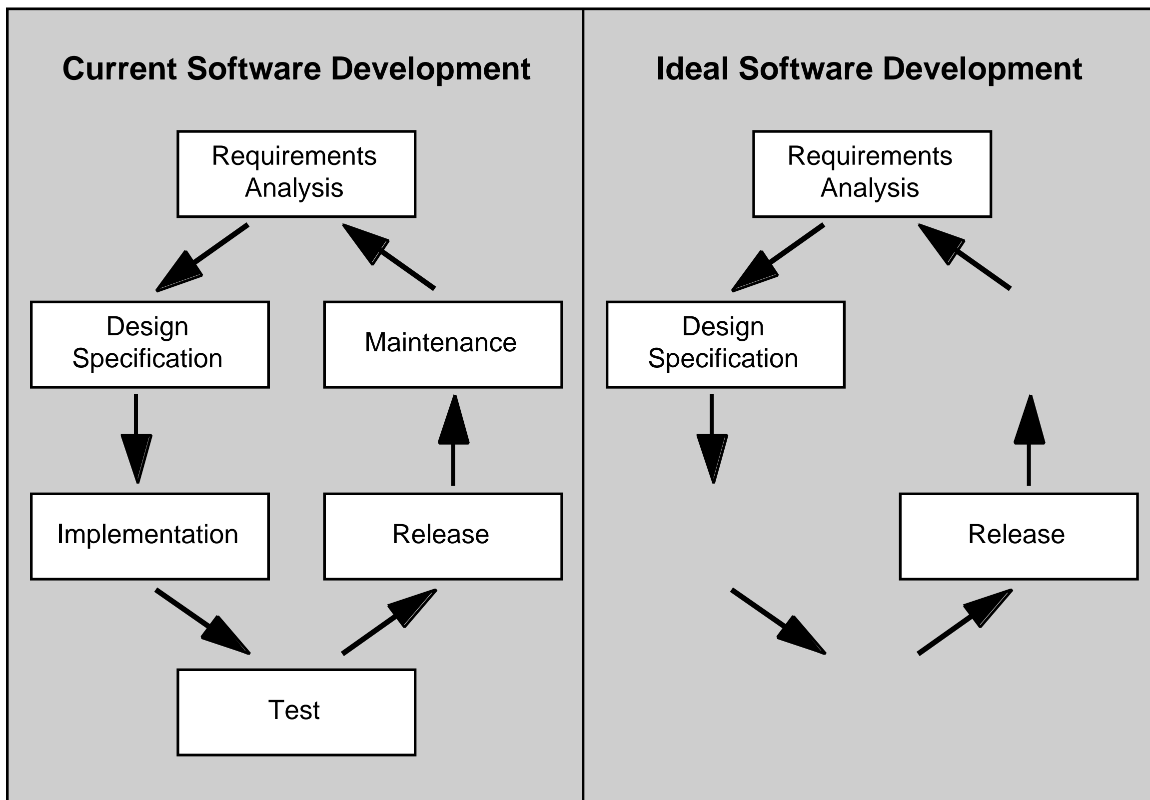


Figure 2.2 Software Development Processes

Software Development in the Aerospace Industry

Software development in the aerospace industry has been greatly influenced by the acquisition management practices of the Department of Defense. These practices include the creation of standards for software development and documentation; such as DOD-STD-2167a, DOD -STD-7935a, DOD -STD-1703, and MIL-STD-498; and the evaluation of the software development capabilities of government contractors. Although recently superseded by the newly created MIL-STD-498, the long-standing DOD-STD-2167a had been the primary standard and has had a significant impact on software development. It specified the process for developing mission critical software for all military systems and defined standards for requirements specifications, traceability and documentation.

Figure 2.3 illustrates the life cycle and structured approach of the DOD-STD-2167a process. The process begins with users and systems analysts defining the system level requirements for a hardware/software system. The process then splits into separate parallel hardware and software development efforts. Hardware and software requirements analyses are conducted to determine partitioning with traceability back to all system requirements. The design phase is broken into preliminary and detailed design. The preliminary design phase defines the software architecture and functionally allocates the software requirements to program modules. The detailed design phase defines the internal specifics of each module and intermodular interfaces. The implementation phase includes coding, debugging and re-coding if necessary. The testing phase involves increasing levels of integration. First individual software modules are tested, followed by the testing of a complete software build. Finally the hardware and software are integrated and tested as a system. The maintenance phase occurs after the system is deployed. This phase primarily involves defining improvements and new system requirements, but may also find errors not detected during testing.

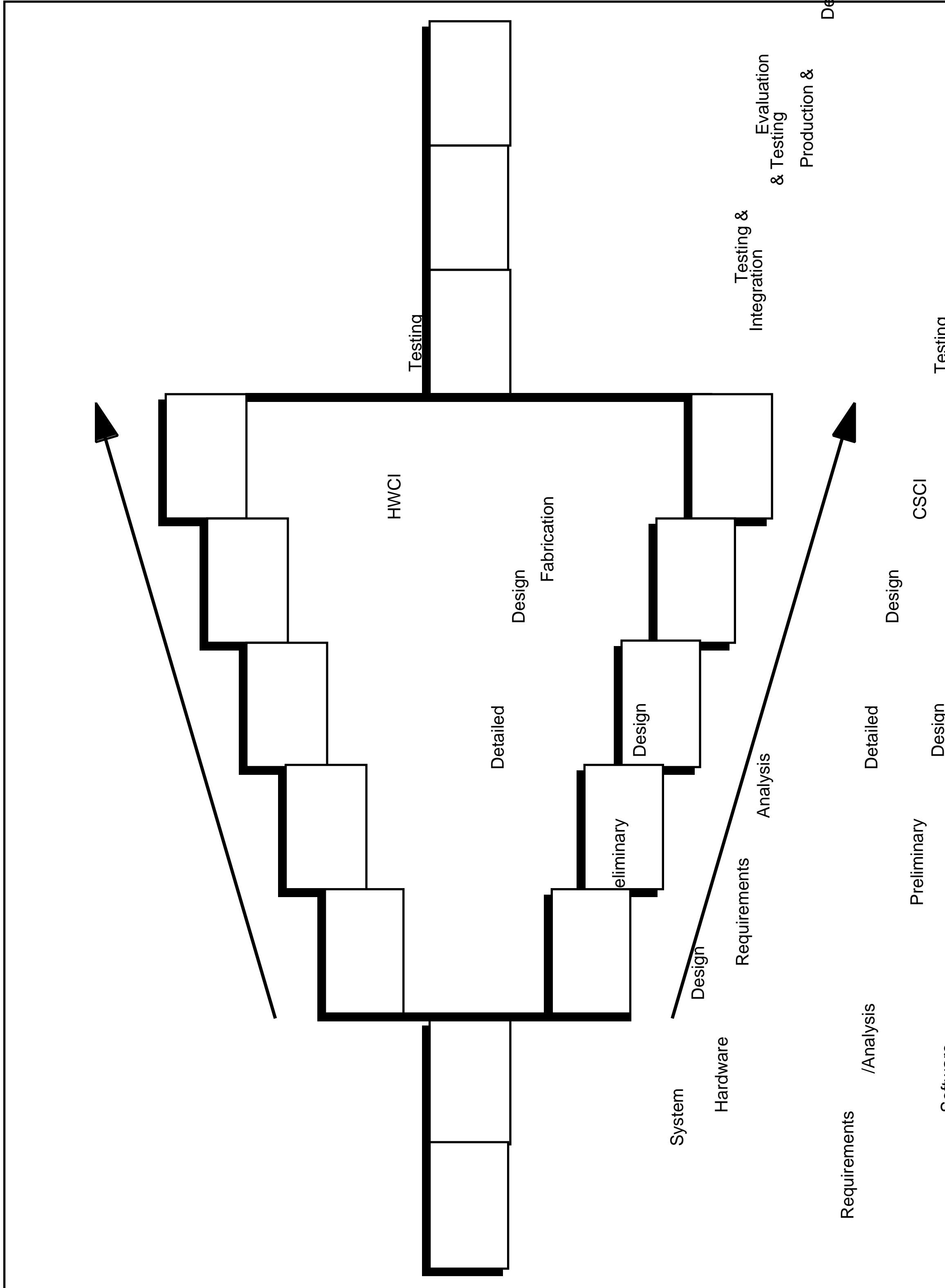


Figure 2.3 DOD-STD-2167a System Development Life Cycle

The DOD-STD-2167a process reflects the classic waterfall life cycle (as shown in Figure 2.3) which implies that each phase of the process is completed once and in sequence. In reality developing software is a very iterative process with many phases occurring simultaneously. Correct and complete requirements are rarely fixed when system development begins. Iteration allows re-evaluation of requirements until operational needs are fully met. Many other life cycle models have been created from the waterfall to capture the iterative nature of the software development process. These include the iterative waterfall, the spiral and the prototyping models. Additionally, new software development processes have been developed such as object-oriented programming which advocates a bottom-up methodology.

The Department of Defense has recently issued MIL-STD-498 to accommodate these new life cycle models and development methods. This new standard specifies requirements for 19 activities, Table 2.1 (Defense Acquisition University, 1996). It emphasizes flexibility by stating that these activities may overlap, can be applied iteratively, can be applied differently to different types of software, and need not be performed in the listed order. MIL-STD-498 is consistent with the acquisition reform initiatives of Defense Secretary William Perry who stated standards should be viewed as guidance instead of strict requirements.

Regardless of which software development process an organization selects, the organization must have a mature management capability in order to repeatedly adhere to the process. The Software Engineering Institute has created a framework for describing an organization’s process capability called the Capability Maturity Model for Software (CMM). The CMM includes five maturity levels which describe a path from ad hoc, chaotic processes to mature, disciplined software processes.

Table 2.1 MIL-STD-498 Software Development Activities

1. Project Planning and Oversight
2. Establishing a Software Development Environment
3. System Requirements Analysis
4. System Design
5. Software Requirements Analysis
6. Software Design
7. Software Implementation and Unit Testing
8. Unit Integration and Testing
9. CSCI Qualification Testing
10. CSCI/HWCI Integration and Testing
11. System Qualification Testing
12. Preparing for Software Use
13. Preparing for Software Transition
14. Software Configuration Management
15. Software Product Evaluation
16. Software Quality Assurance
17. Corrective Action
18. Joint Technical and Management Reviews
19. Other Activities

The five maturity levels of the CMM are:

1) Initial: The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

2) Repeatable: Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3) Defined: The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.
4) Managed: Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5) Optimizing: Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.[2]

The CMM is used as a map for either a software process assessment or a software capability evaluation to appraise the maturity of an organization's execution of the software process. Software process assessments are used to determine the state of an organization's current software process, to determine the high-priority software process-related issues facing an organization, and to obtain the organizational support for software process improvement. Software capability evaluations are used by the Department of Defense to identify contractors who are qualified to perform the software work or to monitor the state of the software process used on an existing software effort. Software capability evaluations are typically conducted by independent evaluation teams.

The Software Factory

As long as a software development process is clearly defined, CASE technology can be introduced to enhance it and improve both quality and productivity. The process should reflect the fundamentals of engineering practice. CASE technology can be used to enforce process control, ensuring the process is disciplined and repeatable. CASE

[2] Paulk, Mark C.; et al: The Capability Maturity Model For Software, Version 1.1, Software Engineering Institute Report No. CMU/SEI-93-TR-24, Carnegie Mellon University, Chapter 2.

technology can automate source code generation, eliminating manual errors and reducing the need for debugging. CASE technology can automate labor-intensive tasks, allowing software development by small teams, a reduction in cycle time and a shift of focus to requirements analysis. CASE technology can be used to collect metric data, facilitating process improvement and schedule and cost estimation. But to fully benefit from all the capabilities of CASE technology, extensive integration into a cohesive software factory environment is required.

Keith Bennett compiled a collection of research papers in his book on software engineering environments (Bennett, 1989). In the first part of this book, dedicated to Software Factories, Christer Fernstrom's chapter provides a good introduction to the factory metaphor and the evolution of CASE technology integration towards a Software Factory Support Environment:

> Many development activities have, of course, long been supported by tools, and recent years have seen improvements in increased coverage and in tool quality. But it is still mainly the responsibility of the individual to effectively coordinate the use of tools. As the word 'tool' implies it is an approach to craftsmanship.
>
> The need for a more industrial approach has long been recognized. It has been one of the major driving forces behind the increasingly ambitious efforts to tool integration. An early example of this is the integration of a compiler, and editor and a debugger into a Programming Support Environment. The next step was the inclusion of design tools, version management capabilities, document support, etc. into what is known as Software Development Environments. Now emerging IPSEs (Integrated Project Support Environments), providing total coverage for a whole project, including for example project management support, represent the next step.
>
> A Software Factory represents yet another step. As the word 'factory' implies it aims at a more industrialized approach to software production. In particular, the Software Factory differs from the previous steps in that it represents a 'people centered view', because it is not only a support environment; it also explicitly includes the integration of people and

their 'corporate knowledge'; their organization, their rules and policies, their methods etc.

The inclusion of people into the system under consideration puts a very important interface in focus: the interface between the people part of the factory and the non-people part - the Factory Support Environment (FSE). The effectiveness of the entire factory is crucially dependent on the quality of this interface, because it determines whether the FSE is a true support to the organization.[3]

Integrating CASE Technology into a Factory Support Environment

CASE technology includes both individual CASE tools and integrated CASE environments. There are two nonexclusive approaches to improving software quality and productivity, a micro-management approach and a macro-management approach (Sage and Palmer, 1990). A micro-management approach attempts to achieve incremental improvements in the various phases of a software development process but leaves the overall process unchanged. A macro-management approach is systemic and wholistic, attempting to achieve improvements by addressing the software development process in its entirety. Implementing individual CASE tools is a micro-management approach that frequently fails to achieve the desired results. Implementing integrated CASE environments, such as a Factory Support Environment, is a macro-management approach. Integration is key to achieving significant improvements in quality and productivity. Figure 2.4 shows the four important forms of integration required to create a fully integrated CASE environment - management, process, team and tool (Bell and Sharon, 1995).

Management integration refers to using CASE tools to control and monitor software development. The tools should collect cost, productivity and quality metrics to allow management of schedules and budgets and facilitate improvement initiatives.

---

[3] Bennett, Keith (ed.): Software Engineering Environments: Research and Practice, Ellis Horwood Ltd., 1989, p. 18.

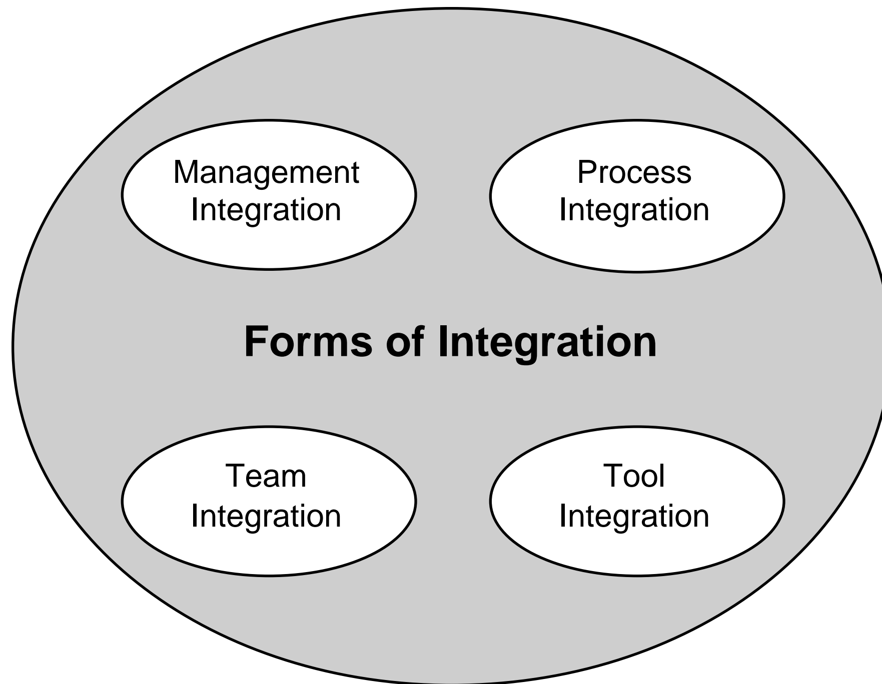


Figure 2.4 Forms of CASE Integration

Process integration refers to using CASE tools that collaborate specifically to support the process and span the process life cycle. The process should not only identify the phases and tasks of software development but it should also identify the tools for each task and the sequence of their use. The tools should allow software developers to follow the process effectively and provide for configuration control.

Team integration, or organizational integration refers to integrating the Factory Support Environment and the organization. For example, tools can be used to electronically network software development teams. The tools should provide for communication and shared access to data while maintaining data integrity. Team integration provides for the "people centered view" of a software factory.

Tool integration refers to using CASE tools that share user-interfaces, data and functionality. The next section discusses tool integration in more detail.

The essential capabilities of an integrated CASE environment needed to capture the four forms of integration are process management, project management, requirements

management, configuration management, document management, a repository, and project verification and validation. In addition the environment should be flexible, extendible and capable of supporting an organization working on multiple projects.

CASE Tool Integration

Of the four forms of integration, tool integration has been the subject of greatest focus. Tool integration addresses the mechanisms of linking individual CASE tools together. There are three dimensions of tool integration - user-interface or presentation integration, data integration and control integration. User-interface integration refers to using a common look and feel among various CASE tools to facilitate ease of use and quick learning. Data integration refers to the representation, conversion and exchange of data in a common standard. It determines to what degree data generated by one tool can be accessed and understood by another. Control integration refers to tool invocation, shared functionality, and the ease of communications between tools. Data and control integration are closely related because tools share functionality by exchanging control messages that contain data and data references. Tools are called interoperable when full data and control integration exists between them.

Figure 2.5 shows three levels of tool integration (Bell and Sharon, 1995). Which level is achieved depends on how the tools were developed. At level 1, individually developed tools are used. They are likely to have common user interfaces and data import/export formats but are unable to share a single repository. At level 2, tools developed together as a suite are used. They are tightly integrated and optimized amongst themselves but their integration with other tools remain at a level 1 capability. At level 3, tools developed to meet formal interoperability standards such as ANSI X3H6 are used. These tools may have been developed individually but together they are capable of forming a highly integrated environment.

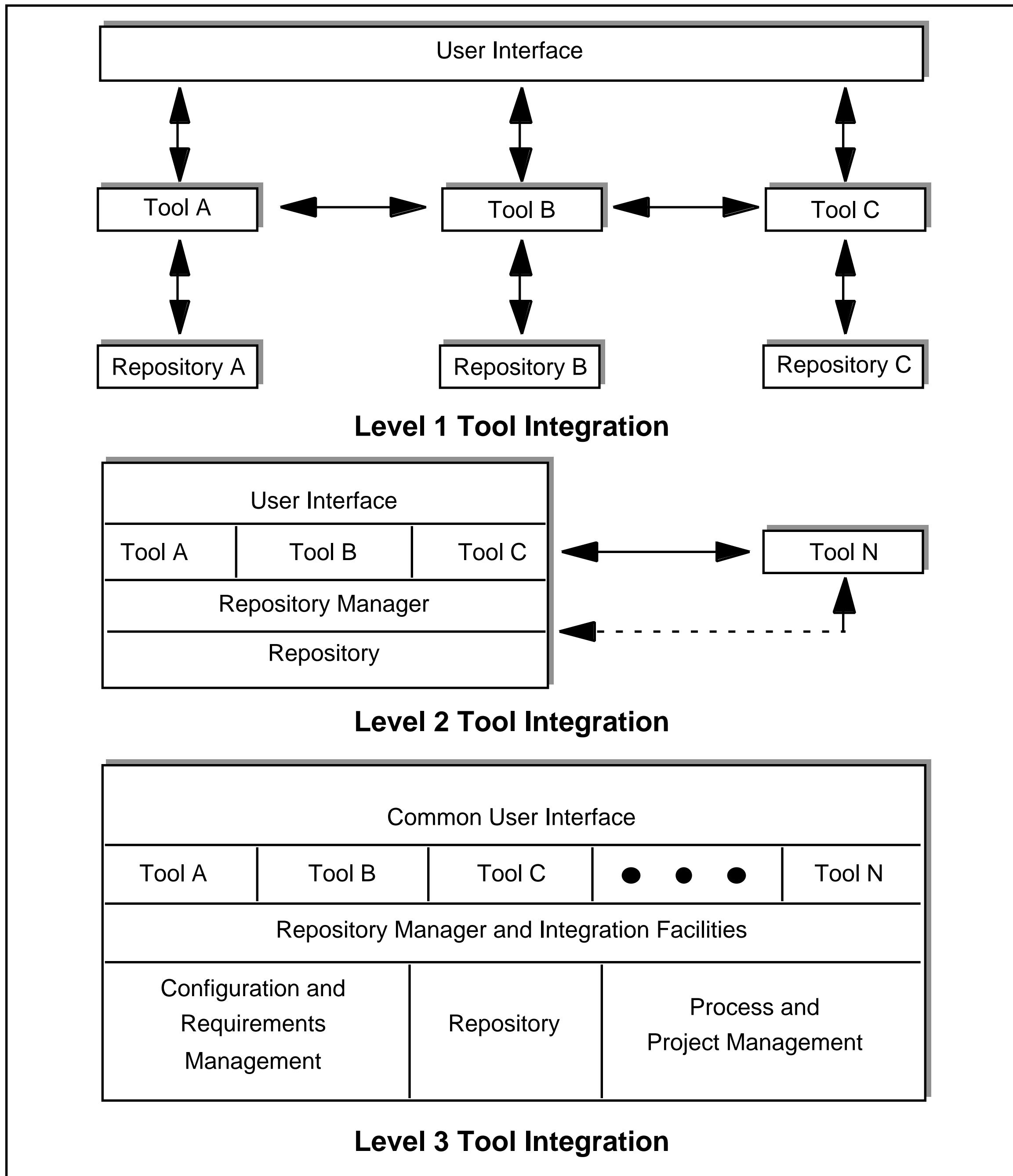


Figure 2.5 Levels of CASE Tool Integration

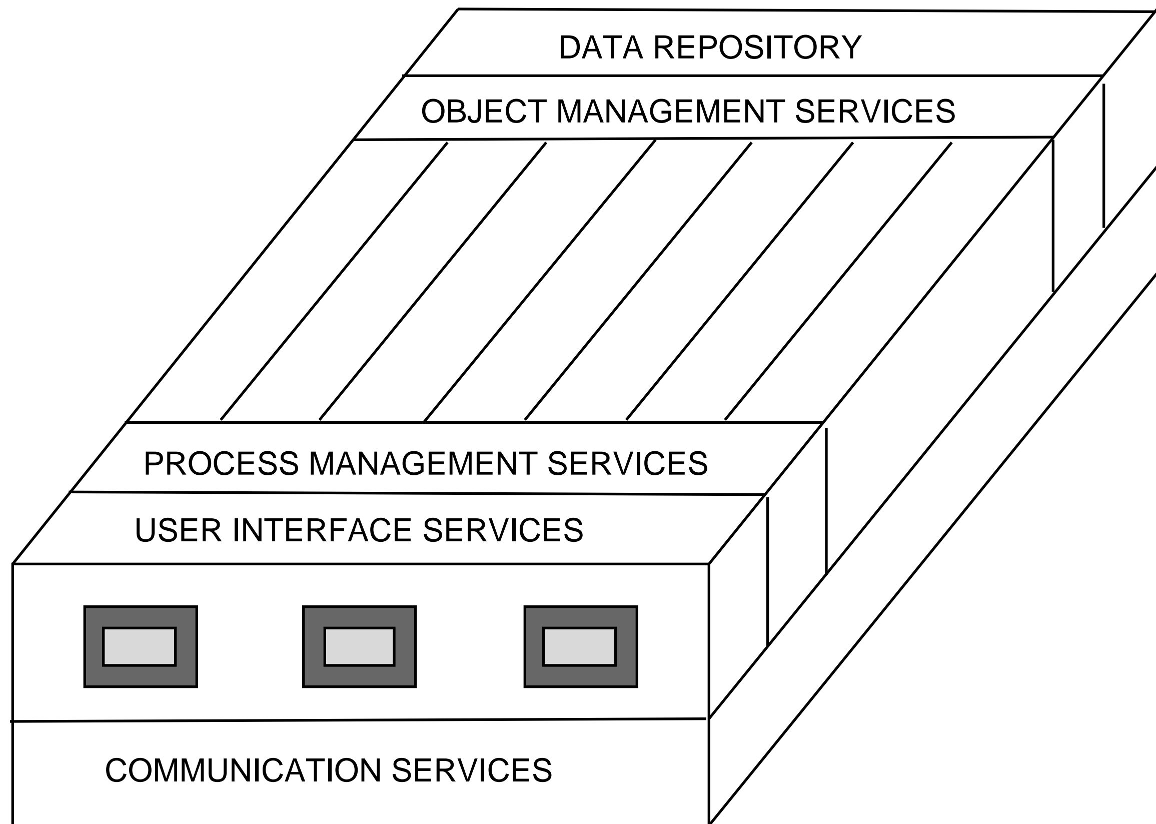


Figure 2.6 NIST/ECMA Reference Model

The "toaster" model, Figure 2.6, represents another form of level 3 tool integration. The "toaster model" is a reference model for an integrated environment developed by the National Institute of Standards and Technology in conjunction with the Europe Computer Manufacturers Association. The model is not an implementation but rather a conceptual framework that includes a catalog of services an environment could provide (NIST/ECMA, 1993). One goal of the model is to be able to define points at which interoperability standards may be useful.

Automatic Code Generation

Any integrated CASE environment must support strong code generation. Without it true process integration and the ideal software development process of Figure 2.2 cannot be achieved. Figure 2.7 shows the evolution of CASE technology leading to automatic code generation (Fisher, 1988). CASE technologies fall into two general groups, Front-End CASE and Back-End CASE (Schefstrom, 1993). Front-End CASE, or Upper CASE, includes graphical notations, editing tools, and other technology developed to support the early requirements and design phases of the software development life cycle. Back-End CASE, or Lower CASE, includes compilers, debuggers, and other technology developed to support the later phases of the life cycle. Figure 2.8 illustrates how automatic code generation bridges the gap between these two groupings of CASE technology by automating the implementation phase while leveraging existing technologies like compilers and graphical editors. Automatic code generators must be integrated into any environment if that environment is to support the full life cycle of software development.

Automatic code generation improves both quality and productivity. Producing defect free source code eliminates implementation errors which are one-third of the errors affecting weapon system development, Figure 2.1. Automating the coding process, a labor intensive task, is necessary to increase productivity. In fact employing CASE technology that delivers a design only, requiring manual implementation of code, can potentially reduce productivity. Employing CASE technology, with full automatic code generation, can create an environment that will increase productivity and reduce errors.

> Automatic code generation is the ultimate goal of most CASE tool vendors and certainly of all CASE tool users. Code generation is the ability to automatically generate working or compilable software directly from a

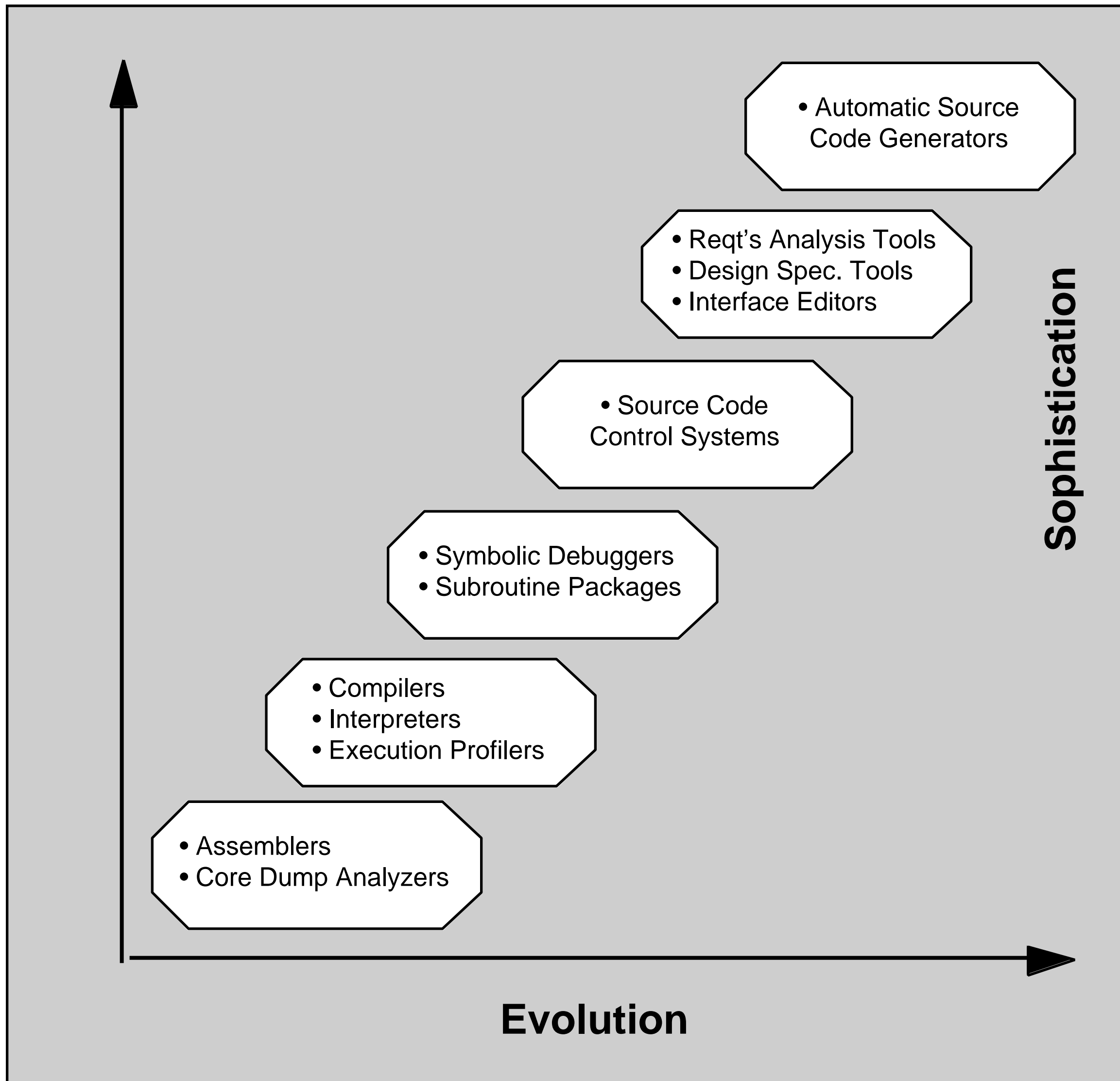


Figure 2.7 Evolution of CASE Technology

design specification. Ultimately, the software designer's time is much better invested in fleshing out application specifications and architectures rather than in coding and debugging. Unfortunately truly generalized code generation is not available but code generation is available in a variety of focused tools. Code generation is a bottleneck in CASE technology.[4]

[4] Fisher, Alan S.: CASE: Using Software Development Tools, Wiley, 1988, pp. 30-31.

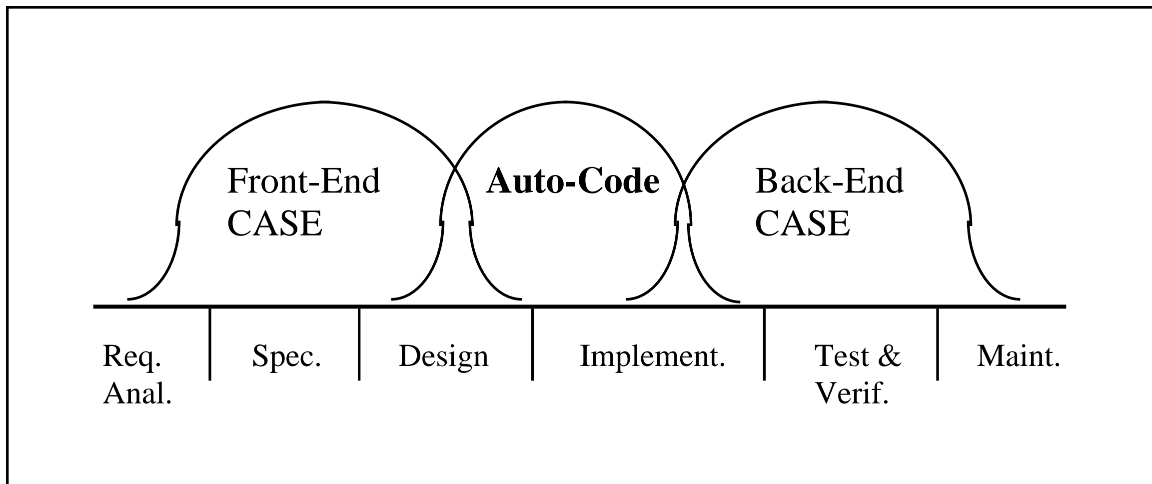


Figure 2.8 Automatic Code Generation bridges the gap in CASE Technology

Years later Fisher's assessment still holds. There is no extensive use of generalized code generation in the defense aircraft industry. However there are several examples of the successful use of automatic code generation to create software for aircraft subsystems, discussed in detail in Chapters 3 and 4. Capable automatic code generators for guidance, navigation and control subsystems exist. Engineers have historically used block type diagrams to specify the detailed design of these subsystems. These code generators employ graphical editors used to create detailed design diagrams and multi-lingual translators used to convert the diagrams into source code. Details of the programming language are hidden from the system designer.

Automatic code generation also improves productivity by facilitating component reuse.

> Component reuse, of course, does not just mean the reuse of code. It is possible to reuse specifications and designs. The potential gains from reusing abstract products of the development process, such as specifications, may be greater than those from reusing code components. Code contains low-level detail which may specialize it to such an extent that

> it cannot be reused. Designs or specifications are more abstract and hence more widely acceptable.[5]

Software factories support component reuse by storing reusable design specifications in a electronic repository. These specifications can be retrieved from within a graphical editor to be incorporated into new applications and then automatically translated into code. If modifications to the specification are necessary, they can be readily created within the graphical editor. Reuse of design specifications with multi-lingual automatic code generators does not confine the application to a predetermined programming language. Also, reuse of design specifications aids portability, allowing the reuse of software across different operating systems and processors.

Strategic Management

Strategic management is an important aspect in any successful attempt to improve quality and productivity and properly implement CASE technology. The problems plaguing software development are interrelated, difficult to separate and inherently systemic. Strategic management involves pursuing integrated solutions by combining organization, process, design, CASE technology and training to complement and reinforce each other.

> This review of problems and solutions in software engineering suggests several observations. Individual engineers and managers know what difficulties they are likely to face, and they have a variety of tools, techniques and concepts to help them. Phases of the development process are interrelated, however, and large projects require many individuals and groups to complete and maintain. Meanwhile, companies continue to face shortages of experienced software personnel and engineers while programming tasks continue to grow in size, complexity, and costs. The result is that discrete solutions, such as in the form of stand-alone tools and

[5] Somerville, Ian: Software Engineering, Addison-Wesley Publishing Co., 1996, p.396.

> techniques, are inadequate to address all the problems organizations face in managing software development.
>
> Boehm and a colleague reinforced this argument in a 1983 article. Their studies of software cost and productivity at TRW, as well as of project performance at other firms, all pointed to the need for comprehensive, strategic efforts on the part of management. They noted that, "Significant productivity gains require an integrated program of initiatives in several areas, including tools, methods, work environments, education, management, personal incentives, and reusability." As for the benefits of such an approach, their studies indicated that, "An integrated software productivity program can produce productivity gains of factors of two in five years and factors of four in tens years, with proper planning and investment." Yet they did not suggest firms could achieve such gains easily, and warned that, "Improving software productivity involves a long, sustained effort and commitment."[6]

Managers must assume a long-term view when strategically implementing CASE technologies. Investment in CASE technology must also be linked with process improvements, research and development, training and organizational changes to form an integrated approach. Also clear quality and productivity goals should be established prior to embarking on any improvement program.

Process analysis and standardization must be completed prior to committing to specific CASE technologies. The process drives tool selection. Case studies have shown that building an integrated CASE environment can take years of effort. In addition to procuring commercially available CASE tools, building an environment may also require in-house development of CASE tools which are specific to the particular domain of application. Although software development frequently displays diseconomies of scale, standardization and a centralized approach to process and CASE technology research

[6] Cusumano, Michael A.: Japan's Software Factory, Oxford University Press, 1991, p. 87.

provide an opportunity to capture economies of scope. Productivity can be enhanced by applying a common process and technology across multiple projects.

Training must include teaching the common process and tool usage to make the development of quality software repeatable and predictable. But an important focus of training is to improve employee skill-sets by also teaching the principles of high-level systems design, requirements analysis, team management and communication. CASE technology can be met with resistance due to the de-skilling nature of automation. Expanding the focus of training beyond tool usage is a strategy for leveraging the front end of the development process.

Automation through CASE technology allows the development of software with a fewer number of highly skilled people. It also breaks barriers of the distinct functional roles between systems and software engineering. This allows the linking of CASE technology with organizational changes such as developing software with small integrated product teams.

Chapter 3: CASE Technology Development in the Aerospace Industry

Numerous development efforts have been underway in the aerospace industry to develop computer-aided software engineering (CASE) technologies that support the creation of software factory support environments. These development efforts involve the creation of integrated environments themselves or enabling technologies like automatic code generation. This chapter provides an overview of some of the development programs and a description of the resulting environments and enabling CASE technologies. Chapter 4 discusses industry experiences using the technologies and summarizes benefits to software development and productivity improvements. Technology development has been underway at various places including aerospace corporations, commercial tool vendors, and government agencies, Table 3.1.

Table 3.1 Environments and CASE Technology in the Aerospace Industry

Company	Environment / Technology
United Technologies	Pictures-to-Code
General Electric	BEACON
Integrated Systems, Inc.	MATIXx Product Family
McDonnell Douglas	RAPIDS
NASA - JSC	Rapid Development Lab
Lockheed	LEAP
Honeywell	DSSA Toolset
Draper Labs	CSDL CASE System
Verilog	SAO + SAGA

Table 3.1 also outlines the order of this chapter. The information presented for United Technologies is based on numerous visits to Pratt & Whitney Commercial Engines and Hamilton Standard. The information presented for the remaining companies is compiled from published reports and articles obtained through library research. Additional visits and interviews will be conducted to obtain further information, which will be incorporated into a future revision of this report. This chapter closes by summarizing the common characteristics shared by the developed technologies.

United Technologies / Pictures-to-Code

United Technologies applied an integrated approach to improving software development through Pictures-to-Code. Pictures-to-Code is a standardized development process and integrated toolset used to create real-time embedded software for control systems. With Pictures-to-Code, United Technologies has been able to decrease cycle time by approximately 40% while reducing errors by more than 60%. United Technologies is now using the Pictures-to-Code process and toolset on 33 programs at Pratt and Whitney, Hamilton Standard and Sikorsky.

The Pictures-to-Code Process

In 1990, the creation of a controls software process improvement team was formed as a strategic decision to reduce software development cost by 50% and reduce software development time by 50%. The team included representatives from Hamilton Standard, Pratt and Whitney Government Engineering, Pratt and Whitney Commercial Engineering and Pratt and Whitney Canada. The primary software products of these divisions are embedded real-time programs for aircraft engine controls, environmental controls and fuel controls. Later their efforts were expanded to include flight controls at Sikorsky.

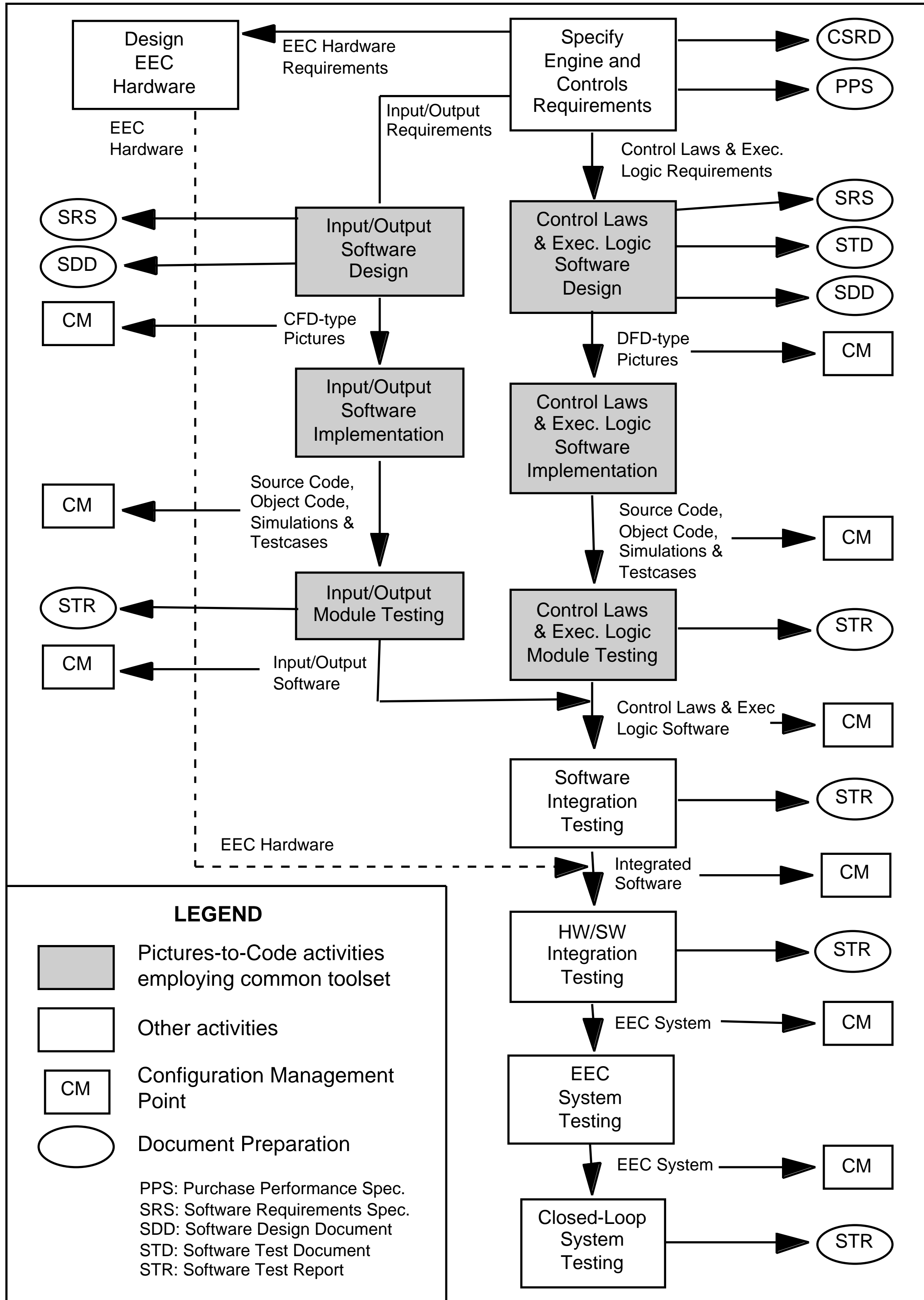


Figure 3.1 Overview of the Common Development Process for EEC Software

At each division the software development was driven by the standards of their various customers including the Department of Defense (DOD-STD-2167A), the Federal Aviation Administration (RTCA/DO-178), Boeing (D6-35071) and the Canadian Department of Transportation. Recognizing that their existing software development practices were all basically similar, but tailored slightly differently to satisfy the standards of their various customers, this team recommended the implementation of a common software development process for both military and commercial programs. This common process would capture economies of scope, facilitate workload balancing and be used across all projects and divisions.

The common process follows the familiar waterfall life cycle. The process steps are iterative with many activities occurring simultaneously. Figure 3.1 illustrates the application of the process to the development of software for an electronic engine control (EEC) system. Constrained by customer standards, the common process does not represent a breakthrough in software development. What is unique, and imposes greater structure to the process, is the extensive use of a CASE toolset created by United Technologies (UTC) to automate and optimize the process.

The Pictures-to-Code Toolset

In 1990 the UTC Pictures-to-Code Working Group was formed to define a common integrated CASE toolset to automate the common software development process. The toolset is common because it is to be used across product lines and divisions. Requirements for the toolset were derived from the common software development process and were:

- database management to archive all objects (software modules, documentation, etc.)
- configuration management to control and document changes to all objects
- automated creation of documentation (requirements specifications, design documents, etc.)

- utilities to create and edit diagrams and text files
- automatic code generator to create source code from diagrams
- compilers, assemblers and linkers
- automated generation of module test cases and command files to execute them
- automated execution of the module test cases and generation of test reports

The working group comparatively evaluated the CASE tools used in each division as well as tools available on the commercial market. This approach made the toolset a synthesis of the best available tools. A preference was given to commercial tools which are continually evaluated for inclusion in the toolset. Figure 3.2 provides a functional view of the Pictures-to-Code toolset.

Automated Code Generation with PtC

The core of the toolset is the automated code generator developed by Hamilton Standard. Hamilton Standard has been involved in controls systems since the 1960's and develops software for engine, flight and environmental controls. The automated code generator is based on a picture language and includes the Graphical Processing Utility (GPU). The picture language consists of block diagram representations used to create software specifications and detailed designs. The automated code generator translates the picture language into a target source code (Ada, C, FORTRAN or Assembly).

The GPU is a graphics editor used to create or alter design diagrams. Each design diagram is a pictorial representation of a separately compilable software module or unit. The GPU automatically creates source code from the design diagram and a data dictionary. The data dictionary defines the data type, ranges, and initialized value for all the variables of the software program and can be accessed from within the GPU. The GPU uses the data dictionary to create the comment header and declarative statements of the source code and the design diagram to create the executable statements of the source code.

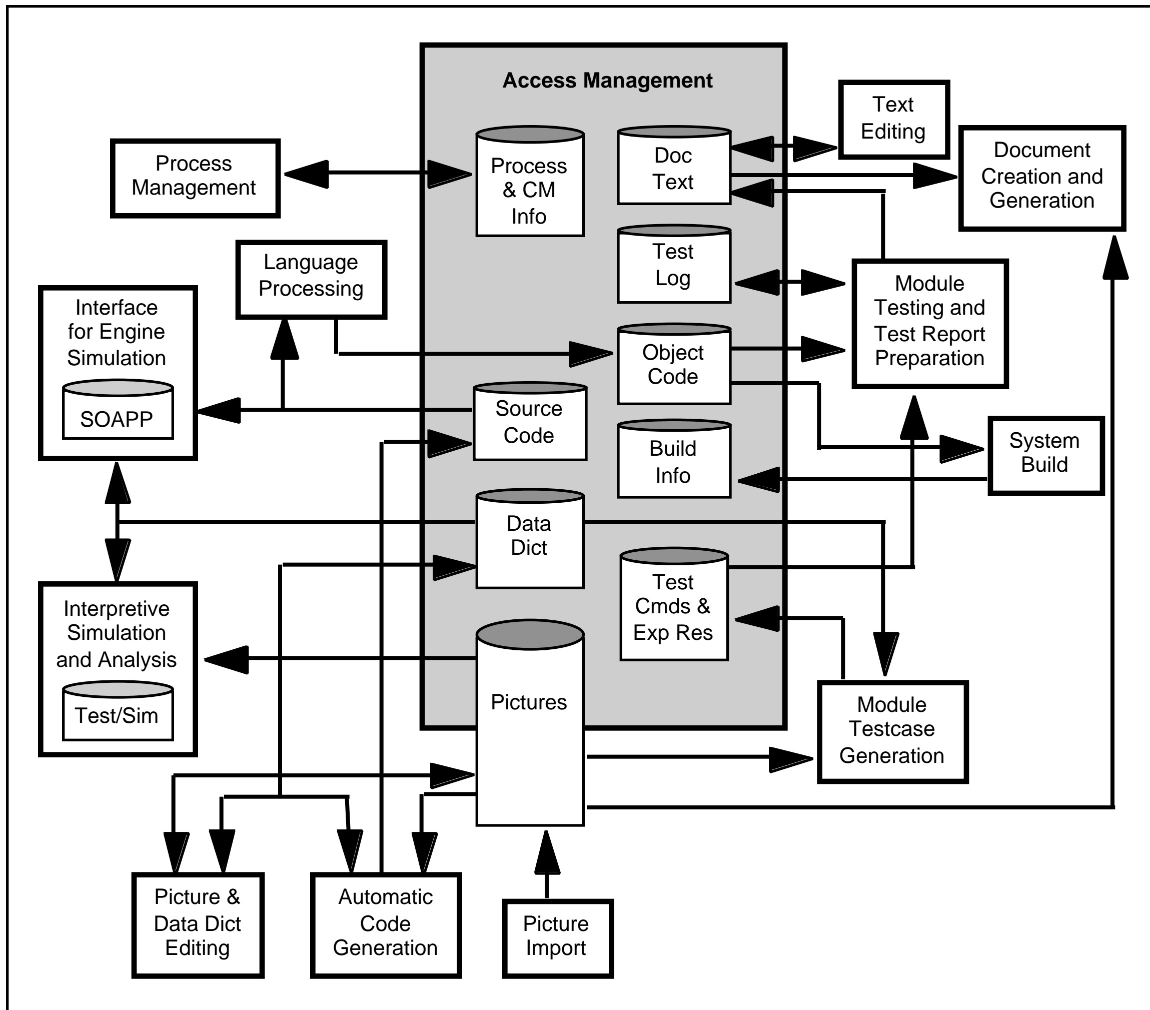
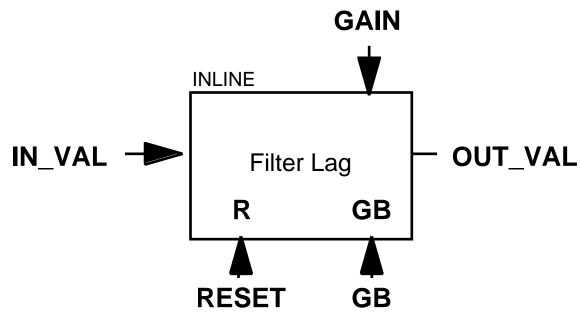


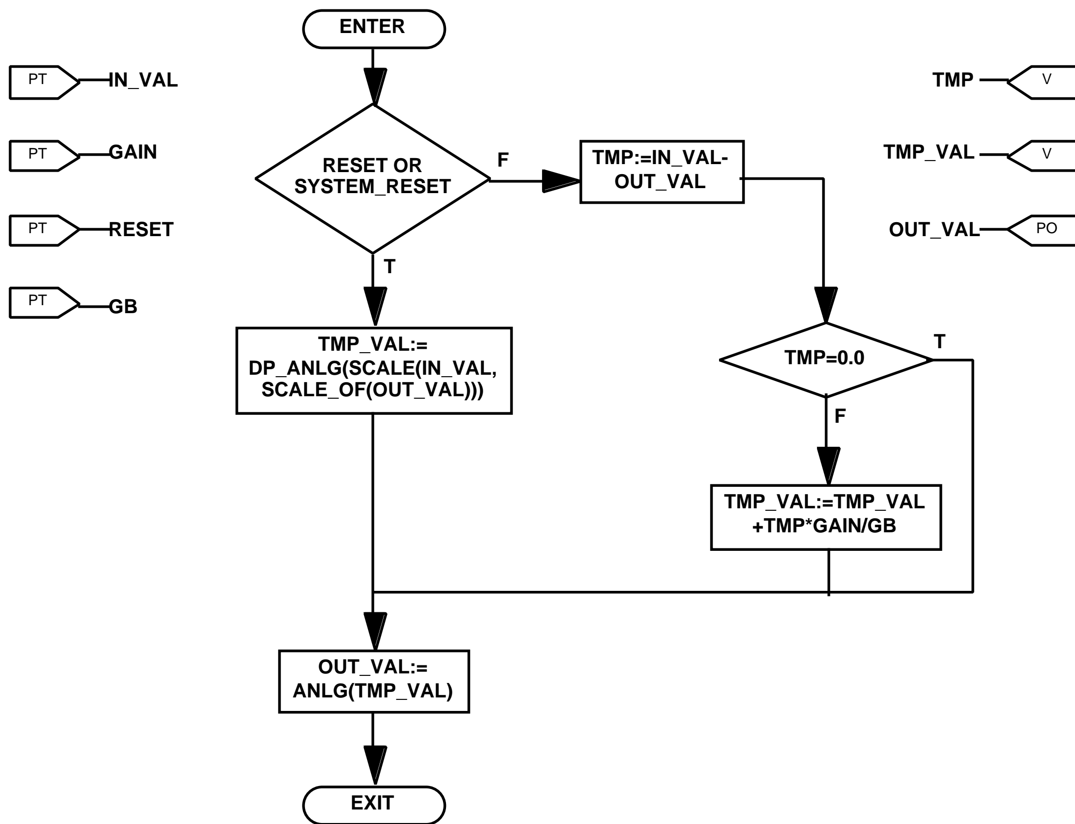
Figure 3.2 Functional View of the Pictures-to-Code Toolset

There are three types of diagrams: Control Flow Diagrams, Data Flow Diagrams and Package Diagrams. Control Flow Diagrams illustrate the execution of several possible sequences of operations (statements including IF, CASE, Loops, etc.). These diagrams require specific entry and exit points. A sample Control Flow Diagram is presented in Figure 3.3. Data Flow Diagrams illustrate the execution of a specific sequence of operations (statements including mathematical expressions, function calls, etc.) and therefore do not require entry and exit points. Data Flow Diagrams are similar to control system block diagrams. A sample Data Flow Diagram is presented in Figure 3.4. Package Diagrams simply illustrate a collection of related Control Flow and/or Data Flow Diagrams.

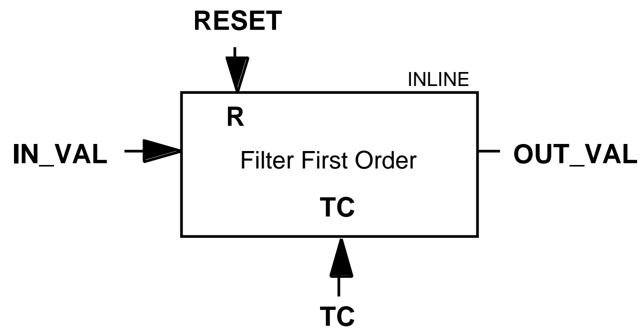
Sample Standard Part:



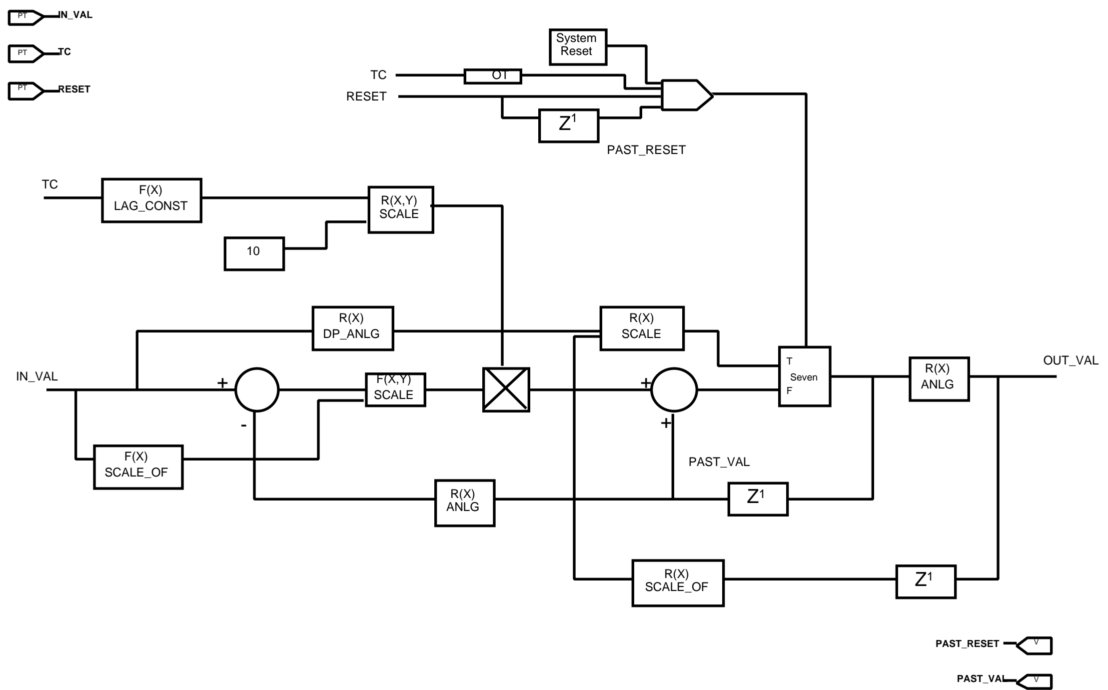
Sample Control Flow Diagram:
(Standard Part Implementation)



Sample Standard Part:



Sample Data Flow Diagram:
(Standard Part Implementation)



The design diagrams are a collection of “primitives” (a shape/figure representing a function) wired together by lines. Each design diagram contains “pins” in the corner which illustrate the interface of the diagram with the Data Dictionary. The picture language is based on a standard library of fifty six data flow primitives and six control flow primitives.

There are three important aspects of the automated code generator: the inclusion of standard parts in the picture language, the inclusion of a compilation/diagnostic capability within GPU, and the final compilation capability of the GPU. Within the picture language a subset of the data flow primitives are called standard parts. These standard parts are functions which are used frequently in the domain of application. The standard parts include proven designs for a counter, timer, integrator, filter, switch and other basic functions which no longer have to be recreated by users. The availability of useful standard parts increases productivity and ensures standardization across software programs. Figures 3.3 and 3.4 also illustrate standard parts.

The GPU has a compilation/diagnostic capability. While in the GPU, a user may request that the diagram created be compiled. This function creates source code from the design diagram. The engineer is notified if the compilation is successful or provided diagnostics if the compilation fails. If there is a failure, the areas of the diagram which caused the problem are automatically highlighted. This capability allows the engineer to stay in the design-check cycle, testing the correctness of the software design before submitting it for review or returning it to the software development library.

The GPU is a true compiler. This means that the diagrams are not simply translated into source code but that multiple primitives on a given diagram can be combined to form a single line of source code for efficiency. Code efficiency is an important software characteristic for real-time embedded systems. While less efficient than the best software engineers, the GPU is at least as proficient in its use of time and memory as a highly experienced software engineer. Occasionally tight tolerances in timing or size limitations require the use of hand-coded software but this only accounts for 5-10% of total code.

General Electric / BEACON

General Electric (GE) Aircraft Controls applies an integrated approach to improving software productivity through BEACON. BEACON is a computer-aided engineering environment used to design, model, and generate code for a complete real-time controller. BEACON stemmed from an independent research and development project started within the GE Aircraft Controls Systems Department in 1984. The goal was to develop an automatic code generation capability and the IRAD project gained the attention of GE Corporate Research and Development. Full scale development of BEACON began in 1989 and was first used to create production code in 1992. GE reports that the use of BEACON has reduced the overall cost and cycle-time for developing aircraft engine controllers.

The BEACON Integrated Environment

BEACON is an integrated environment with the ability to automatically generate code, Figure 3.5 (Rimvall, 1993). The environment supports system design, system analysis and simulation, auto-code generation, system integration and test, and auto-documentation. The major tools of BEACON are a graphical editor and an automatic code generator.

The graphical editor is used to design control systems. The editor has a menu bar, palettes and drawing screen used to create block-diagrams from symbols. The symbols represent some numerical or logical operator. Figure 3.6 presents some sample symbols. There are about 60 symbols allocated to nine palettes. The symbols can be interconnected to form either signal-flow diagrams and control-flow diagrams. These are equivalent in function to the data-flow and control-flow diagrams of United Technologies Pictures-to-Code environment. Diagrams are grouped hierarchically to define the system. The BEACON system also comes with a symbol editor which allows users to develop new symbols.

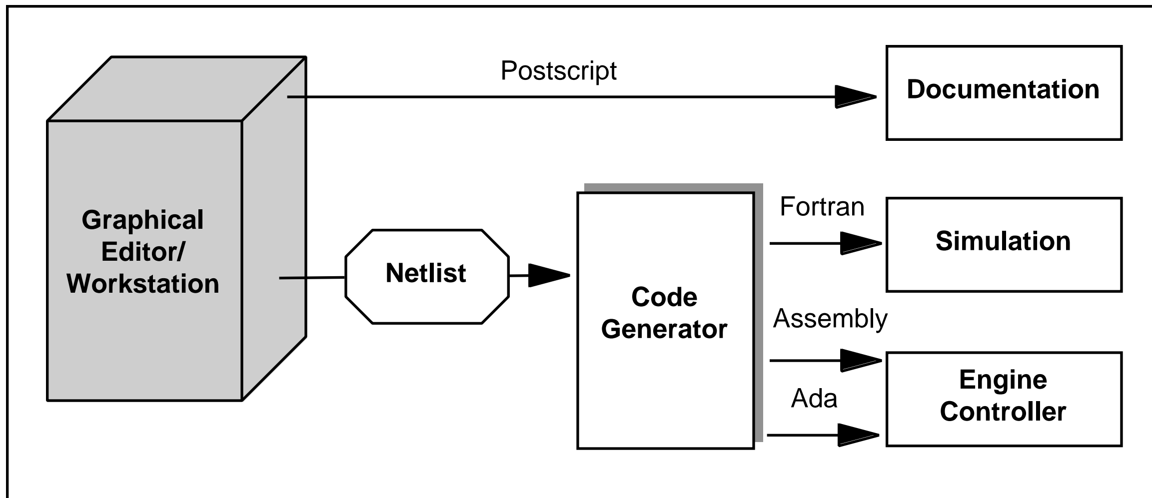


Figure 3.5 The BEACON Environment

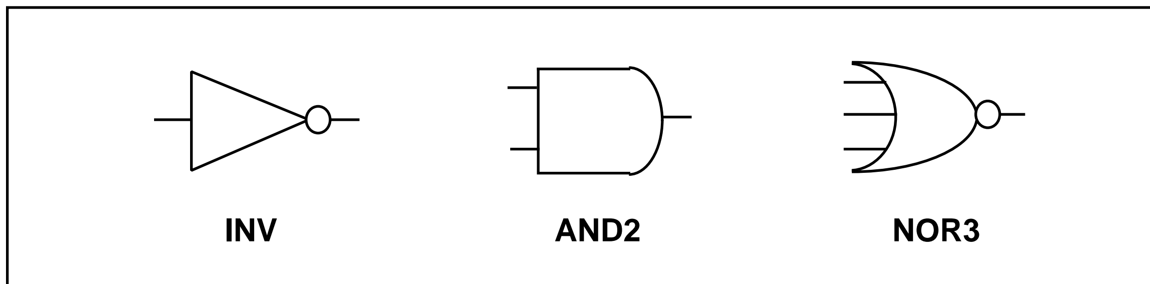


Figure 3.6 Sample Symbols from BEACON

Each symbol, or block, has a corresponding definition described in the BEACON Block-Definition Language (BDL). The definition describes the functionality of the symbol, input/output interfaces, and data declarations. The BDL is a special-purpose structured language which allows the automatic code generation to be independent of the target language. When a diagram is ready to be translated into a target language, the BDL descriptions for each symbol are combined into a netlist file. Prior to creating the netlist, the user selects the check diagram option from the graphical editor menu bar. Warnings are provided if errors are made such as incomplete connections or the failure to define subdiagrams for a hierarchy block.

The netlist file is the input file to the automatic code generator. The automatic code generator translates the system design into source code. The automatic code generator is

The SystemBuild tool is used to create and simulate graphical block diagrams. It allows hierarchical designs, block reuse, and custom block development. Xmath is an analysis tool used to evaluate and display the simulation results from SystemBuild. The AutoCode tool automatically translates the graphical block diagrams into real-time source code. AutoCode can translate into either C or Ada. The DocumentIt tool automatically generates documentation from user customizable templates. It is also compatible with word processing programs like Microsoft Word and Adobe FrameMaker. The RealSim Series (AC-100 Model C40 and AC-100 Model 3) supports processor-in-the-loop testing. The MATRIXx product family can be hosted on a variety of computer platforms.

McDonnell Douglas / RAPIDS

McDonnell Douglas Aerospace West (MDA-West) applies an integrated approach to improving software productivity through the Rapid Prototyping and Integrated Design System (RAPIDS). RAPIDS is an integrated design and development environment used to create real-time embedded software for guidance, navigation and control (GN&C) systems. With RAPIDS, MDA-West claims to have been able to decrease the software design cycle by up to a factor of 10 (Reil, 1993). MDA-West has applied RAPIDS on more than 15 missile and rocket projects to develop GN&C software.

The RAPIDS Software Development Process

The RAPIDS development process features rapid prototyping of a GN&C system from system requirements definition through detailed design and test on a target processor, Figure 3.7 (Riel, 1993). The process is based on the spiral life cycle model and is highly iterative. The total number of cycles for a RAPIDS program is very high, typically greater than 20, with duration decreasing each time and each cycle delivering a complete GN&C system. The duration decreases from months at the start of a program to days at the end when the design is mature.

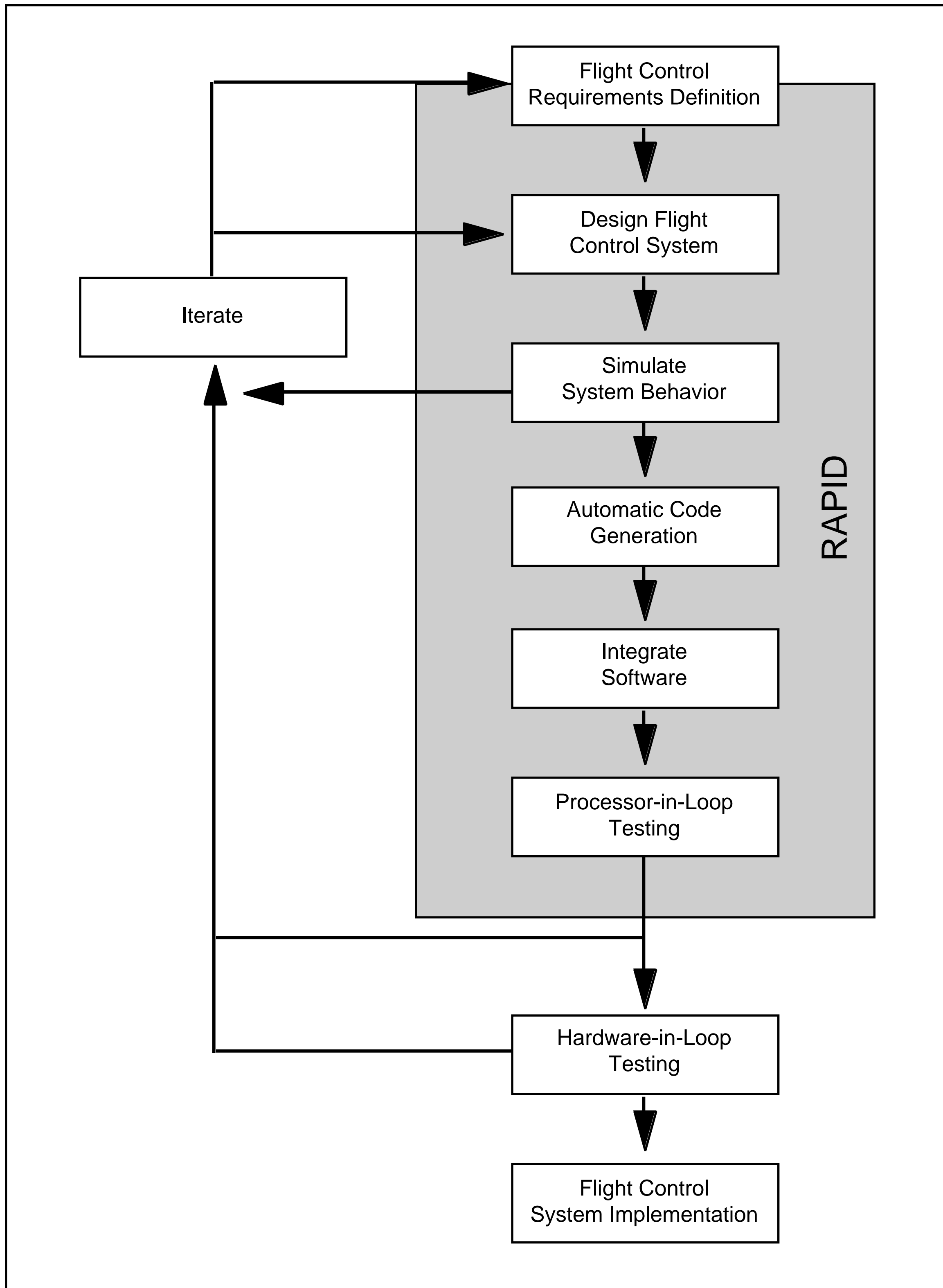


Figure 3.7 The RAPIDS Software Development Process

Rapid development cycles are achieved through the use of graphical design, automatic code generation, reuse of software and simulations, and processor-in-loop (PIL) testing. PIL testing involves running the complete software on a prototype flight processor (R3000, 80960, 1750A, etc.) to get accurate measurements of software memory and timing versus estimations. The highly iterative approach, combined with prototyping, aids in speeding the development cycle by allowing design, implementation and hardware selection problems to be discovered sooner.

The RAPIDS Integrated Environment

RAPIDS features an integrated CASE environment based on a core of commercially available products from Integrated Systems Inc. (MATRIXx/SystemBuild/AutoCode/AC-100) and Cadre (Teamwork), Figure 3.8 (Riel, 1992). The environment allows automated software development from design through test with its capabilities for graphical system design and simulation, automated code generation and processor-in-loop (PIL) prototype testing.

> MATRIXx/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

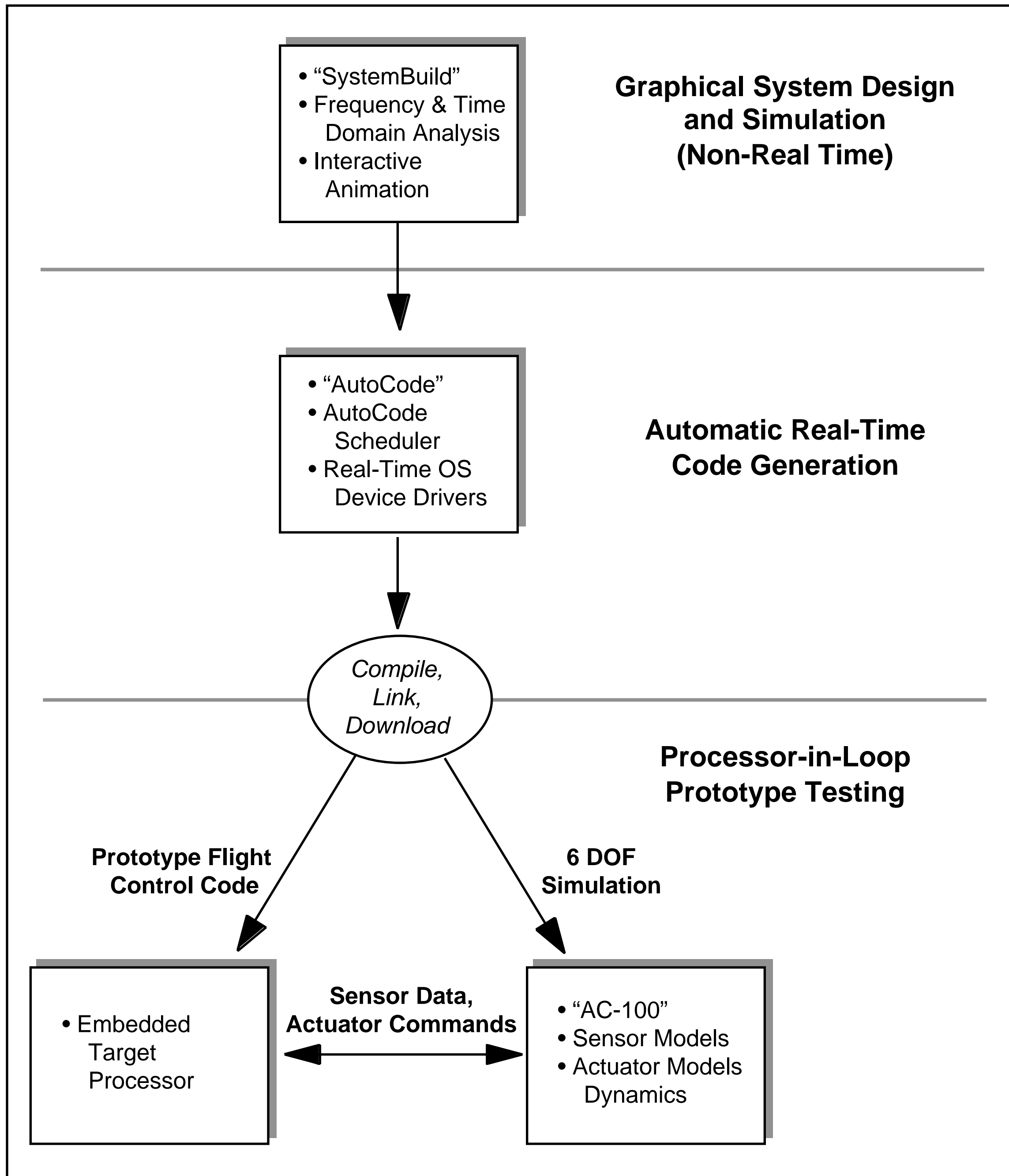


Figure 3.8 The RAPIDS Environment

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.[7]

The environment also includes electronic configuration control, automatically generates documentation and electronically connects integrated product teams over an ethernet network.

NASA-JSC / Rapid Development Lab

The Rapid Development Lab (RDL) is a NASA Johnson Space Center lab created to explore and evaluate new technologies and processes for flight software and simulation development. Recognizing that increases in productivity and decreases in cost and schedule can be achieved through the integration of CASE tools and rapid prototyping techniques and encouraged by the successful application of these practices on such programs as the DC-X and MSTI spacecraft, the Aeroscience and Flight Mechanics Division created the RDL to investigate the uses of these practices on NASA programs. Potential programs for the application of these practices and technologies include the Space Shuttle, Space Station and the Soyuz Assured Crew Return Vehicle. The RDL is a teaming effort between NASA-JSC, Lockheed Engineering Services Corporation, and McDonnell Douglas Aerospace Houston.

The RDL includes the Development Environment and the Real-Time Testing Environment. The Development Environment is based on the MATRIXx product family. The Development Environment is connected to a JSC wide area network allowing remote users to tie-in over the network. Once the system designs are performing satisfactorily, developers proceed to the Real-Time Testing Environment. This environment includes

[7] Hou, Alex C.: Toward Lean Hardware/Software System Development: Evaluation of Selected Complex System Development Methodologies, MIT Lean Aircraft Initiative Report LEAN 95-01, Feb, 1995, p. 21.

high-speed real-time computers, flight-equivalent processors, sensors and effectors, ground support emulation, cockpit modeling, and mission visualization.

NASA used MATRIXx to facilitate software reuse through the construction of libraries (Uhde, 1995). The pilot project of the RDL was a real-time six degree-of-freedom simulation of the crew transfer vehicle for the planned international space station. One of the goals of the project was to create libraries of reusable parts. NASA used MATRIXx "SuperBlocks" to create utility routines. A "SuperBlock" is a collection of elementary blocks interconnected into a data flow diagram. Seven utility libraries were constructed and used to create the simulation. The libraries are comprised of both stand alone utilities and utilities that require some modification by the developer. The libraries are the Vector Matrix Utility Library, the Quaternion Utility Library, the Orbital Element Utility Library, the Plant Utility Library, the Flight Software Utility Library, the Sensors/Effectors Utility Library, and the Miscellaneous Utilities Library.

Lockheed / LEAP

The Software Technology Center of the Lockheed Palo Alto Research Lab created The Lockheed Environment for Automatic Programming (LEAP). LEAP is a synthesis system for automatic code generation linked to a simulation capability. LEAP allows software development from stored components. Lockheed has validated the technology of LEAP on several internal projects over several years, such as programming an autonomous underwater vehicle, but experience remains limited in comparison to the environments previously discussed in this chapter.

The LEAP Integrated Environment

The LEAP environment is centered around a growing library of component templates, Figure 3.9 (Ogata, 1991). The templates are detailed implementations of system components in CIDL. CIDL is the Common Intermediate Design Language developed by

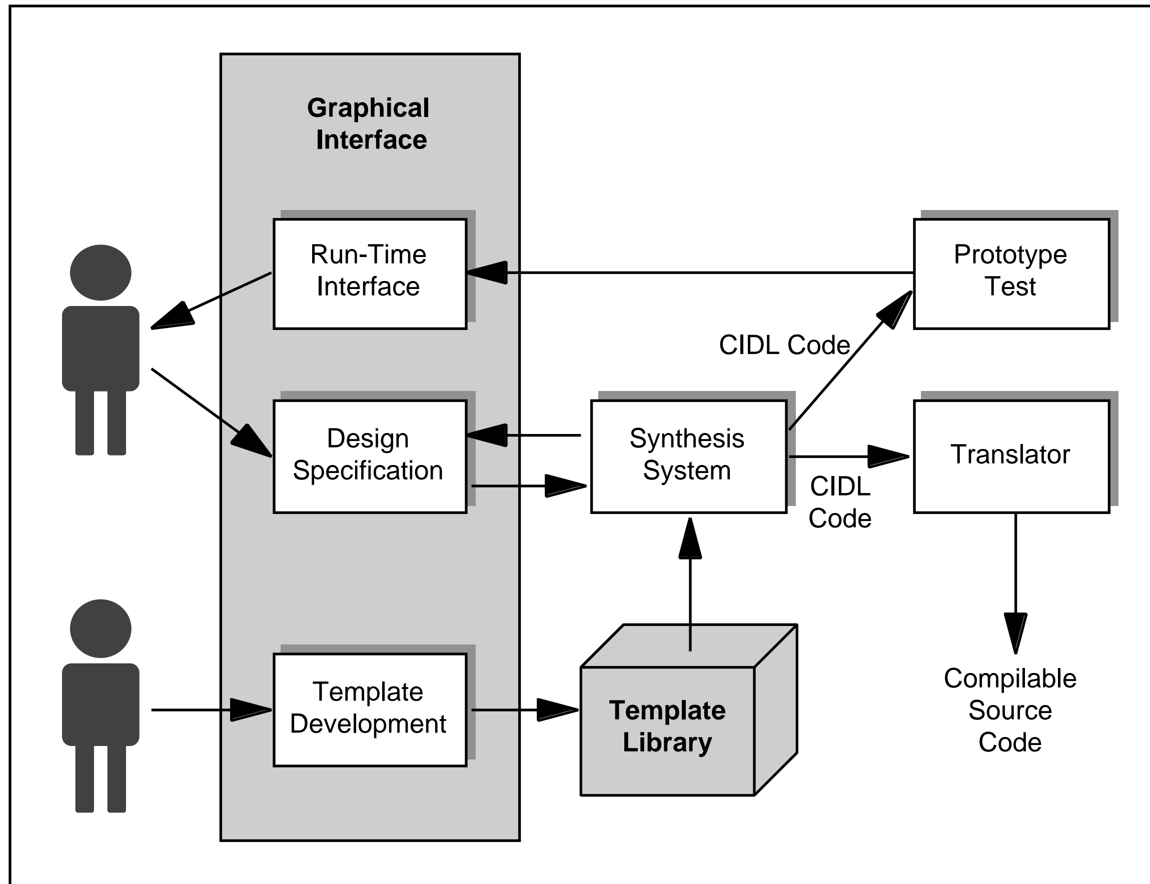


Figure 3.9 Lockheed Environment for Automatic Programming

Lockheed. The LEAP environment includes a graphical and textual editor, a library, a synthesizer and a translator. The synthesizer creates CIDL code from graphical diagrams and vice-versa. The translator creates source code from the CIDL code.

An engineer interactively develops a system description by hierarchically arranging components within the editor. Reusable CIDL components taken from the library are converted to their graphical form by the synthesizer for display. The engineer can modify these components if necessary or create new ones. Once the system is fully described graphically the synthesizer creates a CIDL implementation of the system. Synthesis involves both reusing existing CIDL code and created new CIDL code. If desired, this implementation can be stored into the library. CIDL is an executable language. Testing of

the CIDL code is supported by a Run-Time Interface (RTI). The RTI takes data created by the executing code and displays it in both plot and table forms.

Once the system design is finalized the CIDL translator is invoked. The CIDL translator translates the CIDL code into the desired programming language (Ada, C or LISP). Documentation can be automatically generated from the graphical descriptions, implementation, or test results.

Lockheed implemented LEAP CIDL to simplify the synthesis process. CIDL incorporates object-oriented features, higher order functions, polymorphism, and concurrent constructions. The features of CIDL extend beyond those of programming languages like Ada to include higher order constructs and formal specifications. These additional features facilitate the synthesis of CIDL code from the graphical representation of the system. The translator analyzes the CIDL code and eliminates the higher order constructs when creating source code.

Honeywell and the Domain Specific Software Architecture Program

The Domain Specific Software Architecture (DSSA) Program is sponsored by the Defense Advanced Research Projects Agency. The DSSA Program aims to improve the engineering activities comprising system design by transforming the relationship between system and software engineers. The DSSA program consists of six independent projects which cover a wide range of software development activities. One project includes the development of a toolset by the Honeywell Technology Center to support automatic code generation for guidance, navigation and control (GN&C) systems. These tools have been distributed to industry under research licenses for evaluation and will become commercially available.

The DSSA Toolset

Honeywell developed an integrated toolset to support software development which includes DoME, ControlH, MetaH and EdgH. The toolset is centered around the ControlH code generation tool. ControlH consists of a specification language and a translator.

The ControlH specification language is both textual and graphical and is tailored to the domain of GN&C systems. The specification language is similar to a programming language but more concise. It supports four types of units: operators (primitive and hierarchical), processes, global constants blocks, and global variable blocks. Primitive operators are the building blocks of the specification. AND and ADD are examples of a primitive logical operator and a primitive arithmetic operator, respectively. A hierarchical operator is a collection of interconnected primitive operators or other hierarchical operators. A process is a component of the specification to be run at a fixed processing rate. A process is created by merely declaring its name and computational period. ControlH also supports traditional flow control statements, such as loops and conditionals.

The translator is multi-lingual, creating source code in either C and Ada from a ControlH text file. When translating the ControlH specification, the user may specify whether the source code will be used by the MetaH tool or if it will be referenced by the executive of an external application. MetaH is a companion tool used to build a complete application by combining source code modules and a run-time kernel (closely resembling an executive) into an executable image. MetaH includes a specification language for describing software architectures as well as tools to support schedulability and reliability analyses through the simulation of the MetaH specification. Like the ControlH language, the MetaH language is both textual and graphical.

DoME and EdgH complete the toolset. DoME is a graphical editing tool that allows the user to manipulate the ControlH and MetaH specification languages in their graphical form. DoME allows the user to develop GN&C software in an object oriented environment. When linked with ControlH and MetaH, the user can create ControlH

specifications, source code modules, MetaH specifications, or complete executable images from within DoME. EdgH supports real-time hardware-in-the-loop testing. It is a tool that creates simulated input and output devices that can be included in a MetaH specified software architecture.

Charles Stark Draper Laboratory / CSDL CASE System

The Charles Stark Draper Laboratory (CSDL) developed the computer-aided software engineering (CASE) system as an independent research and development program with the support of the NASA Langley Research Center. NASA support was provided as part of the Advance Launch System (ALS) Development Program from 1988 to 1989 and the system was originally known as ALS CASE. After the termination of the ALS program the system was renamed CSDL CASE and development continued under the Draper Laboratory Corporate Research program as well as the support of NASA. The goal of this development program was to significantly reduce the cost of developing and maintaining real-time scientific and engineering software. The CSDL CASE system provides for the automated generation of compilable source code and accompanying documentation from functional specifications. NASA sponsorship was eventually withdrawn for budget reasons and development of the CSDL CASE system has been temporarily discontinued.

The CSDL CASE System

The CSDL CASE System consists of a graphical user interface, an automatic software designer, automatic code generators, and an automatic document generator, Figure 3.10 (Jones, 1993). An engineer uses the graphical user interface to create functional specifications in the form of engineering block diagrams.

> The computational aspects of the diagram are referred to as ‘transforms’ since they explicitly transform inputs into outputs with no hidden side effects. The data aspects of a diagram are ‘signals’ that carry

> information from one transform to other transforms. Hierarchies of both transforms and signal types can be built either bottom-up or top-down. For bottom-up design, predefined sets of building blocks for both transforms and signal types are supplied. In the case of transforms, these are called primitive transforms and are comprised of such things as add, subtract, multiply, divide, absolute value, switch, etc. For signal types, these are called predefined types and include integer, float, character, string and boolean. The user also can create his own signal types such as arrays and records. For top-down design, the engineer needs only to specify the input and output characteristics of a transform before using it in a block diagram. The details of the transform's data flow and processing can be deferred until a later time, or a body of existing code can be referenced rather than automatically generating code.[8]

The automatic software designer converts the graphical specification into a generic procedural form. The generic procedural form is a text file and is independent of the final target language. The diagrams are converted into procedures, functions, or in-line code with each transform equating to a statement or a block of statements. The automatic software designer also studies the connectivity of the diagrams in order to insert additional variables if needed and determine which statements should be executed in sequence, in parallel, or conditionally.

The generic procedural form is the input to the automatic code generator. The CSDL CASE system can generate both C and Ada code. The graphical specification is also an input to the Automatic Document Generator. The generator creates both text and graphics for the document. The Automatic Document Generator works in conjunction with a commercial publishing software package to format and print the document.

[8] Jones, Denise; and Turkovich, John; et al: op cit, p. 186.

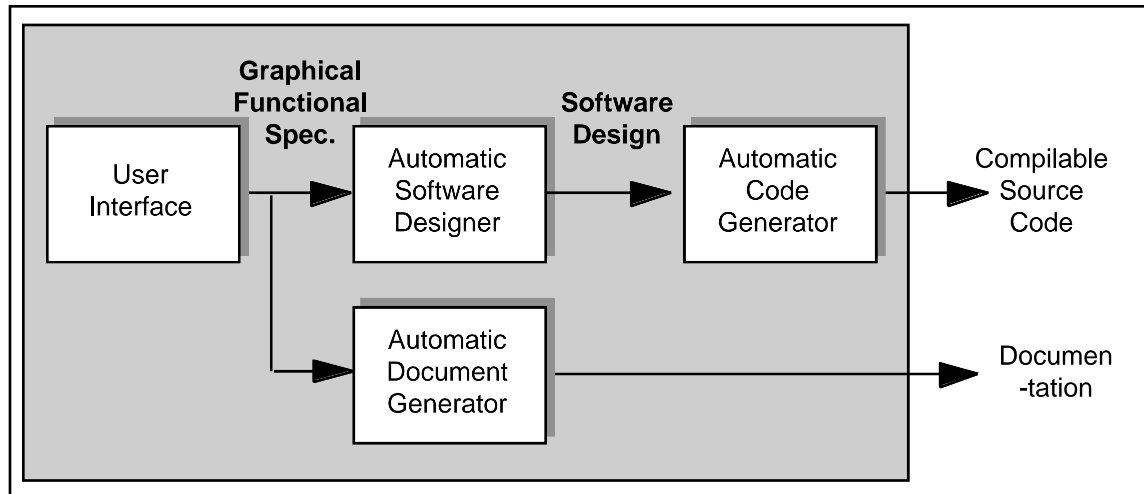


Figure 3.10 CSDL CASE System

Verilog / SAO+SAGA

Efforts to develop integrated environments and automatic code generation technology have not been limited to the US aerospace industry. Verilog, a commercial tool developer, created an integrated environment for designing critical real time systems, the SAO+SAGA Environment. This environment was used by Schneider Electric to create a control and instrumentation system for nuclear power plants and by Aerospatiale to create flight software for the Airbus jetliners. Automatically generated code accounts for 70% of the total on-board software for the A340 jetliner.

The SAO+SAGA Integrated Environment

The main tools of the SAO+SAGA environment are a multiview graphics editor and an automatic code generator. Symbolic design specifications are created in Lustre. Lustre is a synchronous data flow programming language using both block type diagrams and text. The graphics editor is used to create block diagrams describing the system and allows for text editing. Each page of the specification includes a diagram and the inputs and outputs to the diagram. The SAO tool performs automatic checks of consistency between the pages of the specification to ensure that all inputs are available, that all outputs are used and that all

data types are consistent. SAO also includes a configuration management tool, a connection to a library service used to store reusable components, and a documentor used to automatically produce structured documents.

The automatic code generator translates the design specifications into C. The tool allows for optimization of memory size and processing speed. Verilog claims the expansion rate compared to manual coding is only 0.31%. The automatic code generator is connected to additional Verilog tools supporting formal verification of the source code.

Summary

Numerous parallel technology development efforts have been underway for the past six to ten years. These efforts have been aimed at creating automatic code generation as part of an integrated environment for the development of software embedded in real-time systems. This chapter has included brief descriptions of the resulting technologies which display many common characteristics, Table 3.2.

All of the systems employ a graphical user interface. With this interface an engineer creates system specifications from symbols in a mouse driven-menu environment. These specifications are in the form of block diagrams and hierarchical modeling lets them vary from general top-level specifications to detailed functional implementations. Libraries of standard parts or templates facilitate reuse at the design level. The user can interactively check the diagram for correctness before implementing the system in code.

System implementation is handled by automatic code generation, eliminating the need for manual coding. Source code can be generated from within the graphical editors with multiple target languages as options. Many systems also employ an intermediate text language. CIDL and ControlH are a few examples. Having the option of multiple target languages lets the engineer either create the final implementation in Ada or simulate the requirements in a more convenient language, all from the same specification. Finally, the

environments support the automatic creation of documentation which eliminates a time consuming task while enhancing traceability.

Table 3.2 Common Characteristics

- Mouse driven-menu workstation environment
- Block-diagram language for system design
- Graphical editor with diagram checking/warnings
- Multi-lingual automatic code generation
- Capability to create both simulation and real-time code
- Optimizations to create efficient code
- Hierarchical system modeling
- Library for storage of symbols and diagrams
- Data dictionary for parameter definitions
- Automatic creation of documentation

Chapter 4: Industry Experiences with Software Factory Development

This chapter summarizes experiences in the aerospace industry with software factory development. Software factory development is characterized by the extensive use of highly integrated CASE technologies forming an environment supporting process automation, project management, and reuse through electronic repositories of proven design specifications. This chapter builds on the previous one which provided an overview of software factory technologies such as automatic code generation.

United Technologies is a major focus of this chapter. The information presented for United Technologies is based on a case study conducted at Pratt & Whitney and Hamilton Standard. The information presented for the remaining companies is compiled from published reports and articles obtained through library research and a few interviews. Additional visits and interviews will be conducted to obtain further information, which will be incorporated into future revisions of this report.

United Technologies / Pictures-to-Code

The United Technologies Pictures-to-Code Process is a standardized development process and integrated toolset used to create real-time embedded software for control systems. A series of visits to Pratt & Whitney Commercial engines and Hamilton Standard in Hartford, Connecticut formed the basis for an evaluation of automated software development with Pictures-to-Code.

Groundrules and Methodology for Pictures-to-Code Evaluation

Pratt and Whitney (PW) manufactures engines for a variety of civilian and military aircraft. With the introduction of embedded software for electronic engine control (EEC), modern aircraft engines have become integrated hardware/software systems. A case study

was conducted to quantify the benefits of Pictures-to-Code by comparing the development of EEC software by hand-crafted means (Traditional method) and by Pictures-to-Code (PtC method).

The two Pratt and Whitney product families chosen for the comparison are the PW4000 Current family and the PW4000 Growth family. The EEC software for these products are developed by Pratt and Whitney (the engine manufacturer) in conjunction with Hamilton Standard (the EEC hardware manufacturer). The PW4000 Current family includes engines flown on the B747, B767, A300, A310 and MD11 aircraft with the EEC software developed in the Traditional method. The PW4000 Growth family includes engines derived from the PW4000 Current family and flown on the B777 and A330 with nearly 95% of the EEC software developed using the PtC method and toolset.

Metric data collected by Hamilton Standard forms the basis for the comparison. The response variable for cycle time is engineering hours per software module. A surrogate measure was chosen for the quality. This response variable is total errors detected during development per software module. A direct measure of quality, such as total errors detected after product release, was unavailable because the PW4000 Growth engine has only recently entered active revenue service and after this case study was conducted.

The Pictures-to-Code Process

The previous chapter includes an overview of the development of the Pictures-to-Code (PtC) process and toolset as well as a detailed description of the automatic code generation tool. The following revisits the PtC process with added details. Figure 4.1 illustrates the application of the process to the development of software for an electronic engine control (EEC) system. The boxes highlighted in gray signify the steps of the process that employ the PtC toolset. The following summarizes the basic process activities:

- Software Requirements Analysis

  The system-level engine requirements are allocated to the EEC and published in a software requirements specification. Verification is provided by a requirements review. Checklists are completed and a summary report is published.
- Software Design

  In preliminary design, the requirements are functionally divided into modules (units) and published in a software design document. In detailed design the internal operation of each module is defined pictorially and added to the software design document. Verification is provided by a design review. Checklists are completed and a summary report is published.
- Software Code

  Modules are coded in the desired language. Verification is provided by a code read. Checklists are completed and a summary report is published.
- Module Test

  The functional operation is verified. All paths are tested for coverage. Test results are published.
- Integration Tests

  The interfaces between the software modules are verified. Hardware/software interfaces are verified. Test results are published.
- Systems Test

  The operation of the EEC system in a stand alone environment is verified. Test results are published.
- Closed-Loop Testing

  The operation of the EEC system on a closed-loop bench using an actual engine or a sophisticated engine simulation is validated. Test results are published.

Those conducting the verification/validation activities may report any problems with the software by opening a Discrepancy Report. Discrepancy Reports document the type of error found and during which activity. These errors are process errors detected during verification of the software design and are corrected prior to formal release.

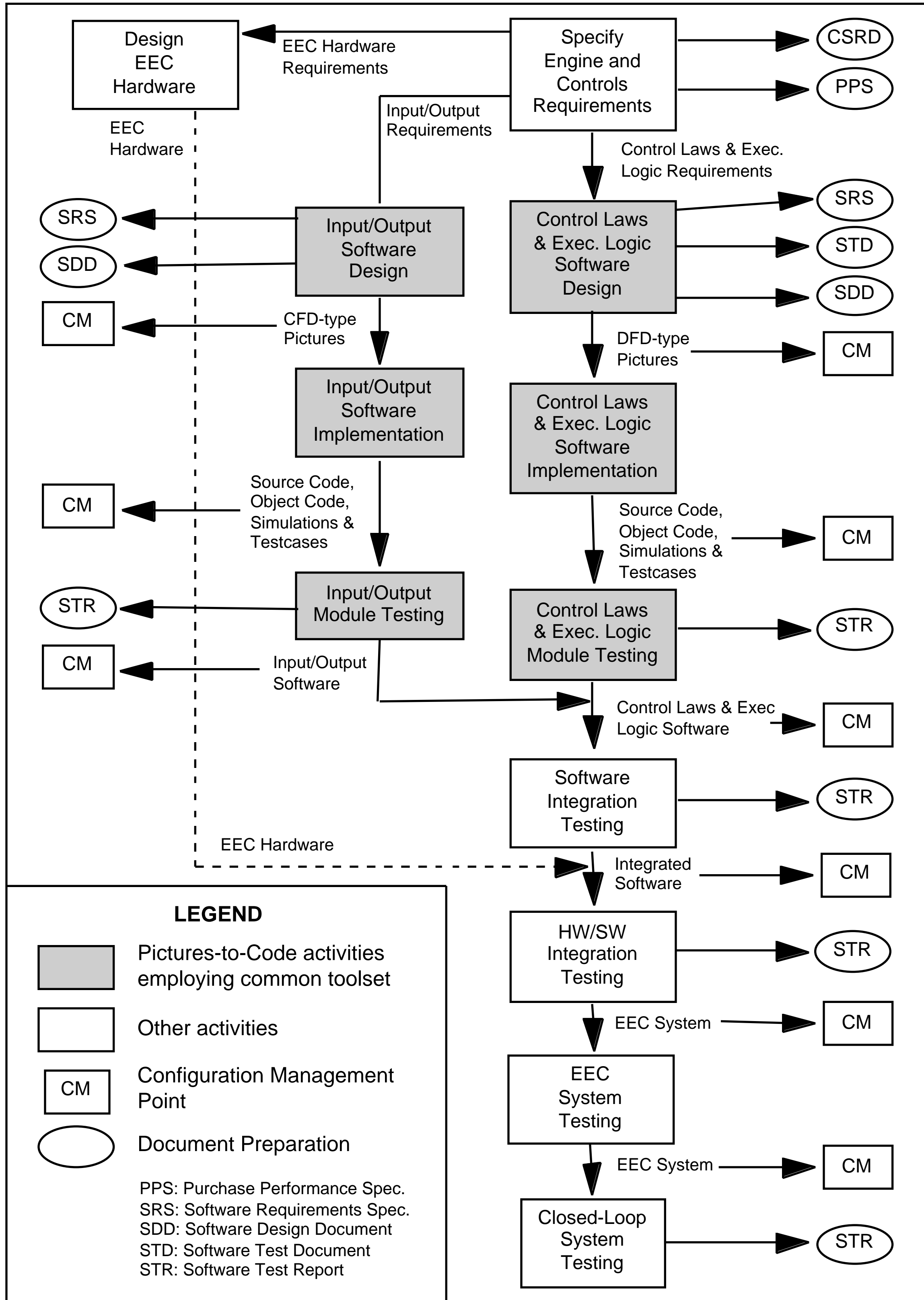


Figure 4.1 Overview of the Common Development Process for EEC Software

Traditional versus Pictures-to-Code

Pratt and Whitney and Hamilton Standard share the development responsibilities for the EEC software. Pratt and Whitney (PW) defines the EEC software requirements based on the Engine Specification from the airframe customer. PW creates the software design for the engine control laws and associated logic and typically delegates the software design of the executive, input/output and maintenance logic to Hamilton Standard (HS). HS programs and compiles all the software designs and links them into a single EEC software program. HS and PW share responsibility for verification of the EEC software while PW is responsible for validation by closed-loop systems testing.

Introduction of the common process and CASE toolset have resulted in significant improvements to cycle time and quality. Hamilton Standard and Pratt and Whitney are now developing software with approximately 60% less errors in 40% the time. A streamlining of the software development process and savings at the software design level are two main reasons for these improvements.

The toolset has streamlined the process by allowing an engineer to function almost exclusively in an automated environment. The engineer has electronic access to design documents, software change request forms and review checklists. These electronic forms can be completed and edited on the computer, reducing the need for paperwork. All formal documentation can be automatically generated by the toolset. Software modules are designed with the Graphical Processing Unit (GPU) and they can be electronically shared between PW and HS. The toolset has also eliminated the need to modify existing source code. To incorporate modifications to existing modules, an engineer alters the module's design diagram and the source code is automatically recreated in its entirety. Source code is not altered by programmers.

Savings at the software design level have also been realized. The software design is still based on structured specifications where the requirements are functionally allocated for implementation but concerns such as the number of source lines of code per module are

removed. Normally a large number of source lines per module are avoided to help programmers retain intellectual control. With the PtC toolset the division of requirements into modules becomes purely functional with no consideration given to the resulting module size. The detailed design requirements for the internal operation of each module is defined pictorially with the GPU. The design diagrams are created by controls engineers with a single diagram becoming a single software module. Engineers focus on the underlying control system design and not the resulting software code.

Since the automated code generator creates source code directly from an engineer's requirements diagram, design interpretations by programmers who may be less familiar with the requirements are avoided. This is reflected in the software design documents. In the Traditional method PW supplies HS the detailed software design through the Purchase Performance Specification. For software requirements, this document contains data information, design diagrams and detailed processing requirements. The detailed processing requirements communicate the intent of an engineer's design to the programmer. The standardized software design document created by the PtC toolset contains only the data information and a single diagram for each module. The detailed processing requirements are not included nor are they needed.

The following figures illustrate the improvements to cycle time and quality realized at Hamilton Standard on large commercial aircraft programs. Figure 4.2 shows a 41% decrease in module cycle time from the 4000 Current (Traditional) to the 4000 Growth (PtC) programs. Cycle time is the time to complete the various activities of software development process and represents the time from receiving a specification change notice (SCN) to delivering a software build to Pratt and Whitney. The values are the total actual engineering hours divided by the total modules altered for all the SCNs received between 1988 and 1994. Figure 4.3 shows an 80% decrease in total detected process errors from the 4000 Current to the 4000 Growth programs. Detected process errors are the number of discrepancy reports opened during the software development process and represent errors

discovered and corrected prior to delivering a software build. Total detected process errors include errors of all types (requirement, design, code, data, documentation). The values presented are the total discrepancy reports divided by the total modules altered for all the SCNs received between 1988 and 1994.

The data presented in the previous figures reflect a small number of SCNs and changed software modules for the 4000 Current program relative to the number of SCNs and changed software modules for the 4000 Growth program. For a further comparison, the expanded information in Figures 4.4 through 4.6 incorporates data from the other large commercial aircraft engine families which used the Traditional method to create EEC software (PW2000 and V2500). Over a period of seven years, Figure 4.4 shows how the data used to create Figures 4.5 and 4.6 reflects an increase in output while the development method transitions from the Traditional to PtC. Figure 4.5 shows that this transition created a trend of decreasing module cycle time. There is a 38% decrease in module cycle time between the first and last years of the period shown. Figure 4.6 shows how this transition results a trend of decreasing detected process errors. There is a 61% decrease in total detected process errors from 0.31 errors/module for the first year to 0.12 errors/module for the last year of the period shown. These improvement trends are attributed to the use of the common software development process and CASE toolset of Pictures-to-Code.

Quality improvement has been shown by examining errors detected during software development because aircraft using 4000 Growth engines have only recently entered active revenue service, however, a corresponding decrease to the errors found in released software can be expected.

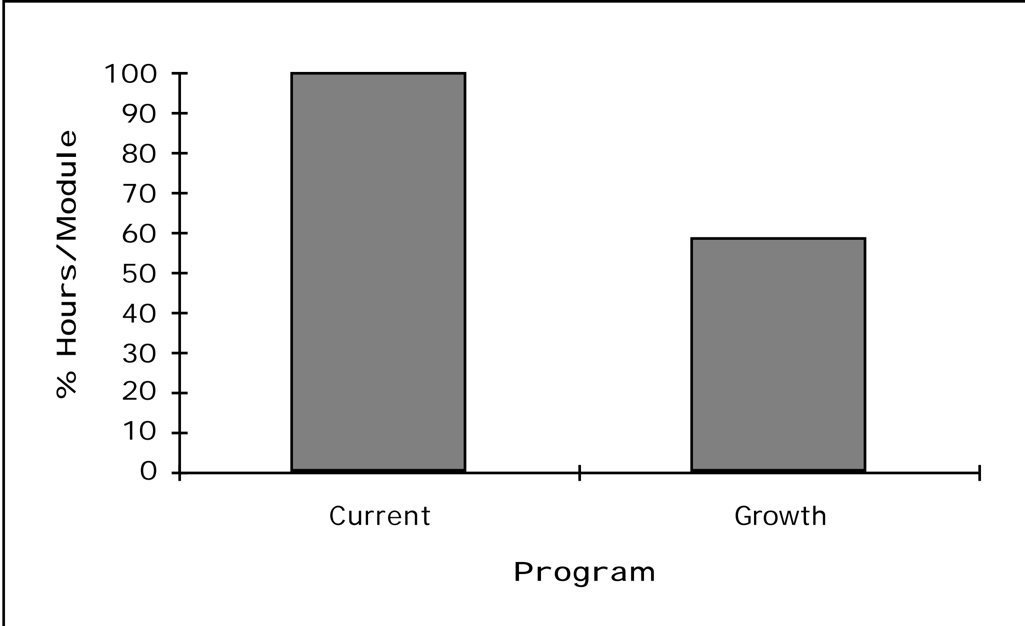


Figure 4.2 % Module Cycle Time, 4000 Current vs. 4000 Growth

(Normalized to 4000 Current)

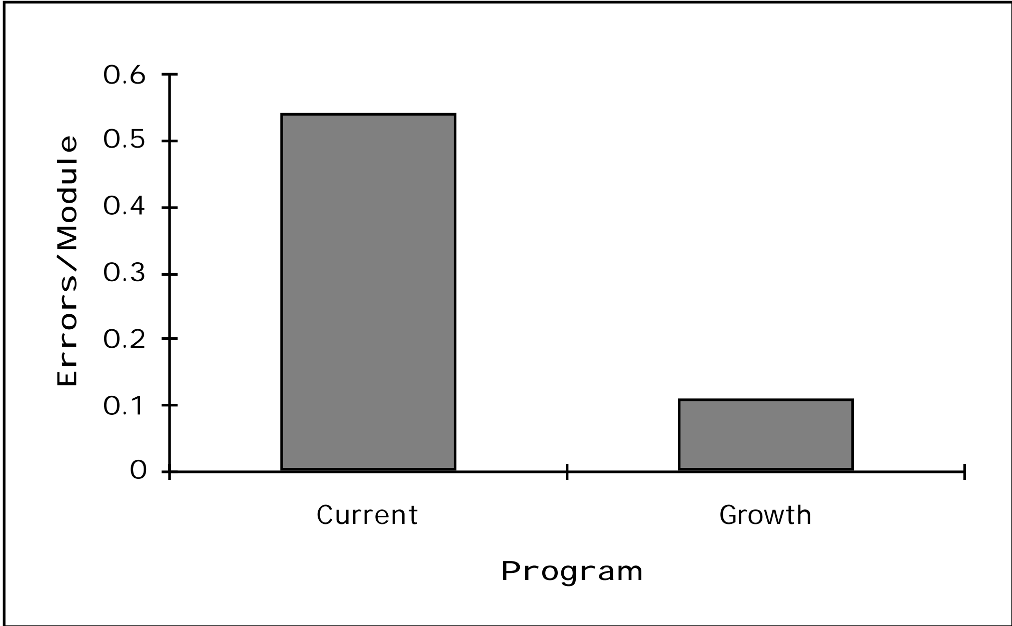


Figure 4.3 Detected Process Errors, 4000 Current vs. 4000 Growth

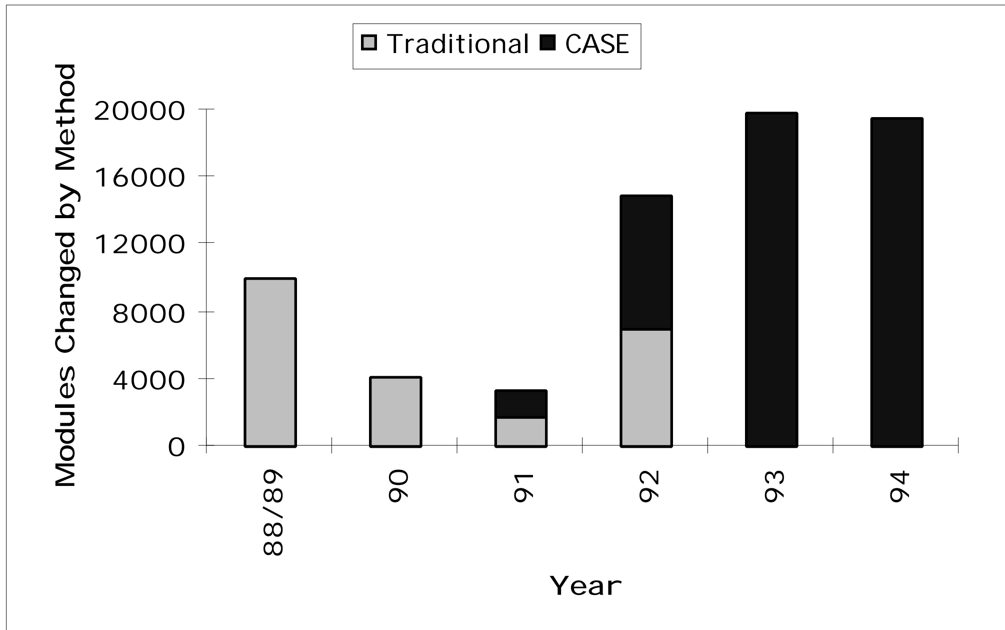


Figure 4.4 Modules Changed by Development Method vs. Year

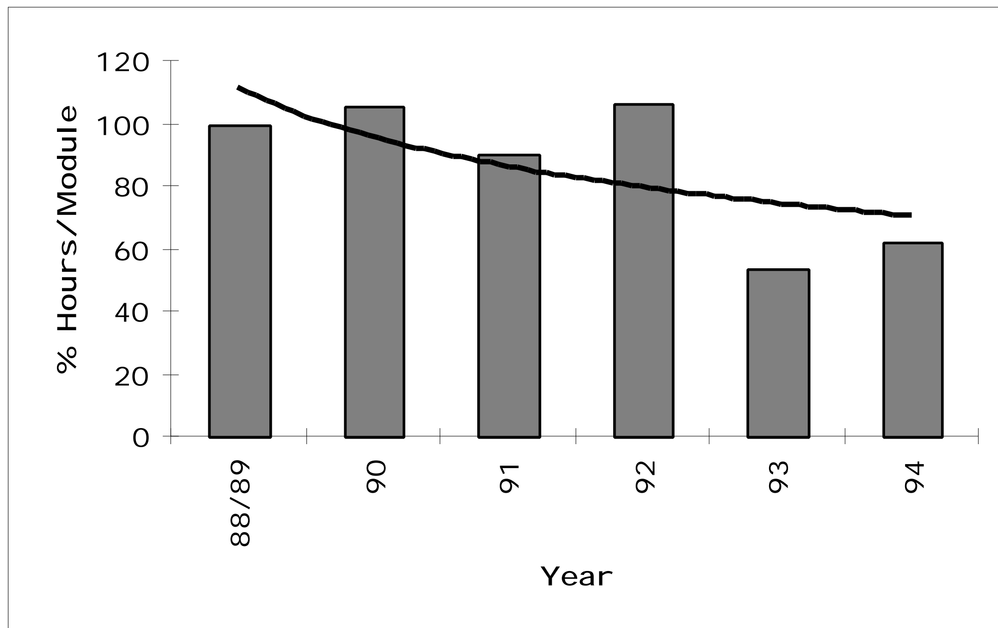


Figure 4.5 % Module Cycle Time vs. Year, All Large Commercial Programs (Normalized to 88/89)

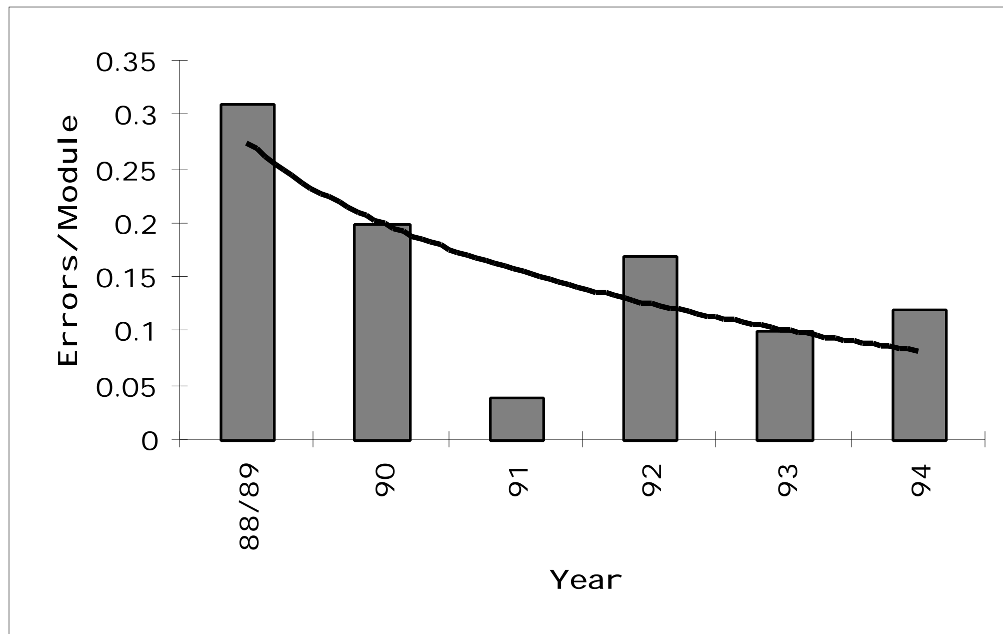


Figure 4.6 Detected Process Errors vs. Year, All Large Commercial Programs

Continuous Process and Toolset Improvement

The development and use of Pictures-to-Code by United Technologies was initiated to reduce the cost of software development. The common software development process drives the toolset and both change with continuous process improvement initiatives. Centered around the automated code generator, the toolset has automated the software development process from top level design through module testing. UTC envisions extending the toolset to encompass the earlier and later activities of the software development process such as requirements analysis, integration testing and systems testing. The toolset is continually evolving with the addition of functionality and enhancements. Any engineer may request modifications or expansions to the toolset. At United Technologies the Software Engineering Process Group is responsible for process improvement and keeps aware of commercially available CASE tools. If superior tools are found they are brought into the toolset. Also, a customer may dictate the use of certain tools other than or in addition to PtC.

The toolset has also identified new areas for improvement in the software development process. Figure 4.7 shows when in the process errors are detected and includes Hamilton Standard data from the 4000 Current and 4000 Growth programs to compare the Traditional and PtC methods. The errors are process errors detected and corrected prior to delivering a software build. The data comes from discrepancy reports as well as checklists completed during the software development process. The figure shows the majority of errors are detected during code read for the Traditional method and that the introduction of automated code generation eliminates code errors which drastically reduces the errors found during code read and module test. This is expected since code reads and modules tests verify the source code versus the software design.

The Pareto diagrams of Figures 4.8 and 4.9 illustrate that the largest number of errors are now system requirements errors detected during systems testing. So with Pictures-to-Code, higher level functional testing is paying attention to true system related problems indicating that software factory methods can aid hardware/software integration. The next step would be to improve system review and simulation to address these system related problems. Also, an improved process would have module testing as the final testing prior to software delivery. Placing integration and systems testing before module testing facilitates more up front attention on requirements.

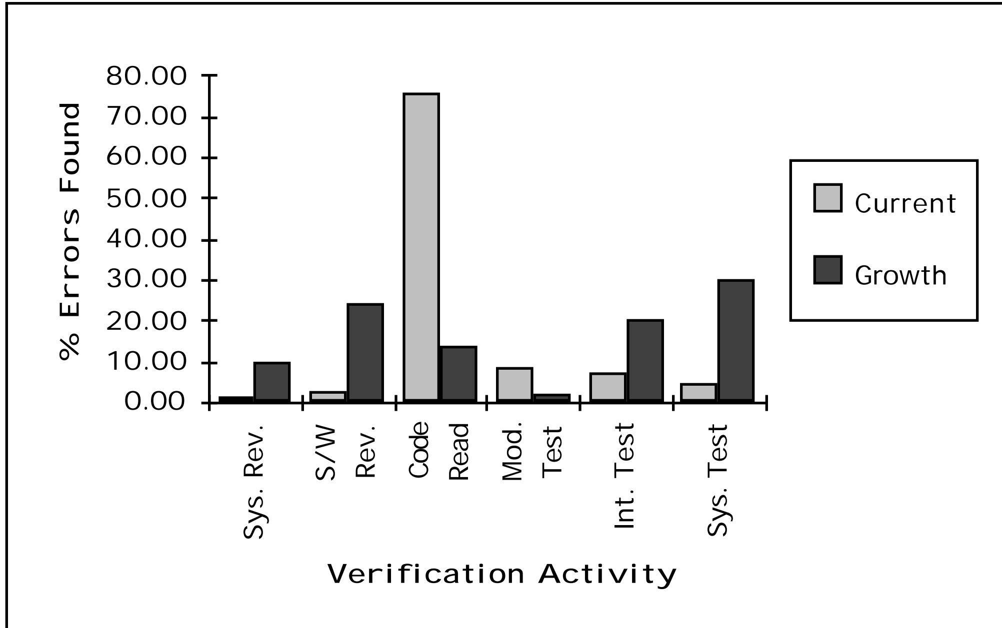


Figure 4.7 % Errors by Program vs. Verification Activity

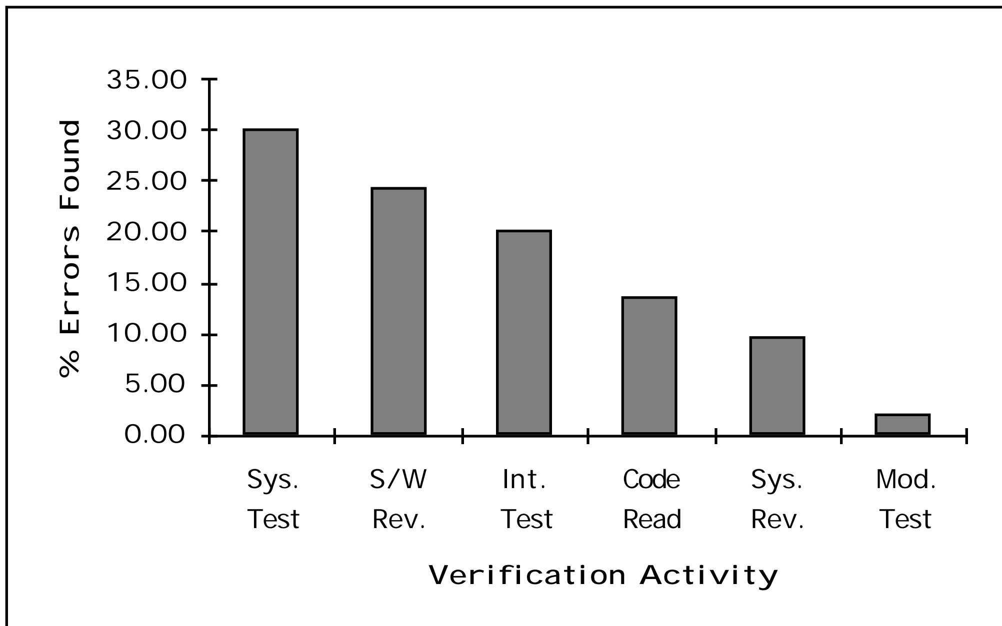


Figure 4.8 % Errors by Verification Activity for PtC Method

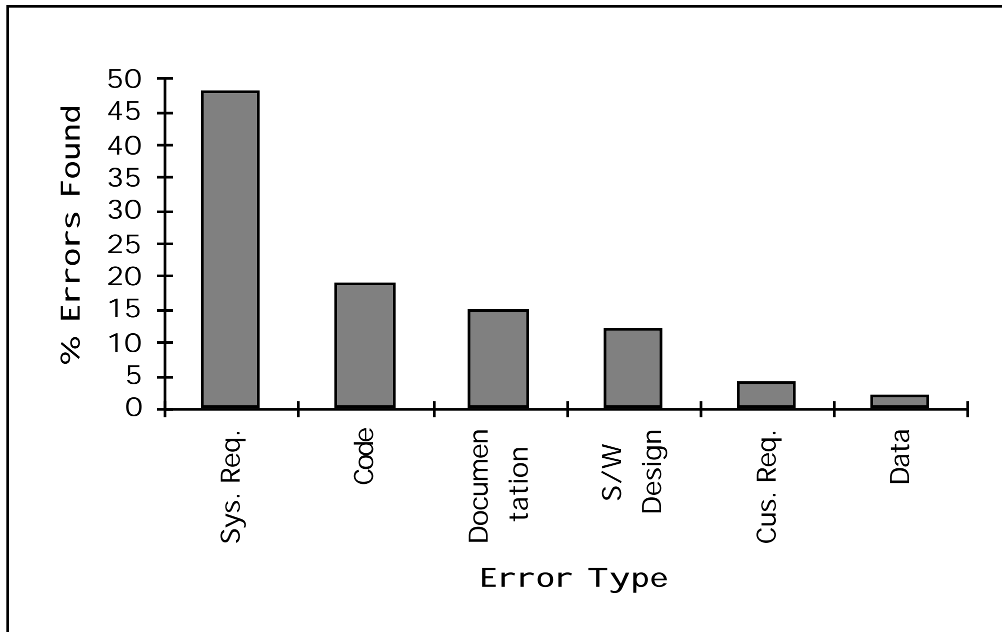


Figure 4.9 % Errors by Type for PtC Method

The Japanese Software Factories

The experience of United Technologies with establishing a common software development process and integrated CASE toolset closely resembles the experience several Japanese software companies have had establishing "software factories" in the 1980s (Cusumano, 1991). While each company had many reasons for starting their "software factories" they all shared the United Technologies' motivations of reducing the cost of software development and increasing the quality of software. These motivations are important as software becomes increasingly complex and a greater amount of system functions are implemented in software instead of hardware. Each company made a strategic decision to capture economies of scope by pursuing a standardized software development process. Then they extensively invested in integrated CASE tools which were gradually introduced over a number of years to automate the process, capture expertise and reinforce good practices. All toolsets included an automated code generator. These management

strategies allowed the companies to capture economies of scope in software development. Long term commitments to process improvement and flexibility were pursued to continually improve productivity and quality and to allow the company to adapt to continuously changing customer needs.

The results for these Japanese companies and United Technologies are highly comparable achievements in software productivity and quality. Tables 4.1 through 4.3 compare metric data for Hamilton Standard against available data for the Japanese "software factories". They all achieved high levels of productivity exceeding one thousand source lines of code (SLOC) per month*. Hamilton Standard also achieved a small and comparable level of detected errors during software development. Although Hamilton Standard has yet to measure the errors for delivered software, numbers similar to those achieved by the Japanese "software factories" can be expected. Also, each company reported a shift of effort from tedious implementation activities such as coding and documentation to the more important design activities as an additional benefit. Table 4.3 shows a breakdown of the software development activities for the various companies.

Table 4.1 Software Productivity

Company	**SLOC/Man-Month**	**Language**
Hamilton Standard	1,378	Ada
Toshiba	1,000	FORTRAN
Fujitsu	1,864	Cobol

* Note: Comparisons must be normalized according to the language used.

Table 4.2 Software Quality

Company	Detected Process Errors (Errors/1000 SLOC)	Detected Product Errors (Errors/1000 SLOC)	Language
Hamilton Standard	1.2	*not available*	Ada
Toshiba	*not available*	0.05-0.2	FORTRAN
Fujitsu	1.5	0.01	Cobol

Table 4.3 % Effort by Software Development Activity

Company	Design	Activity: Implementation	Testing
Hamilton Standard	42%	32%	26%
Hitachi	38%	36%	26%
NEC	47%	37%	15%
Fujitsu	33%	30%	37%

NOTE: Data from Tables 4.1 through 4.3 compiled from Japan's Software Factories.[9]

[9] Cusumano, Michael A.; Japan's Software Factories: A Challenge to U.S. Management, Oxford University Press, 1991, pp. 213, 240-1, 320, 352, 381.

Improving the Productivity and Quality of Software Development

Like United Technologies, all users of automatic code generation report an increase to productivity. Some of the literature describing software factory technology quantified productivity increases for General Electric, McDonnell Douglas, and Aerospatiale:

> A productivity improvement of over 35% has been demonstrated.[10]
>
> Metrics tracked during the DC-X1 software development indicated productivity improvements greater than 25% can be achieved.[11]
>
> Aerospatiale has achieved a productivity gain of 55% in Airbus A320 design tasks.[12]

United Technologies was able to increase productivity by 66%. This measurement is a comparison of data from the PW4000 Current and Growth engine programs based on the productivity metric of modules/hour. Together these yield an average productivity increase of 45%, Figure 4.10.

45% is a significant productivity improvement and even greater improvements may be possible. The amount of productivity improvement is dependent on the initial capability of the organization and the extent software factory practices are implemented. One of NASA's initial demonstrations of the Rapid Development Lab involved creating simulation software for the Soyuz Assured Crew Return Vehicle and a dramatic increase to productivity was achieved.

---

[10] Spang, H. A.; et al: "The BEACON Block-Diagram Environment," Proceedings of the IFAC 12th Triennial World Congress, Vol. 2, 1993, p. 754.

[11] Maras, M. G.; and Riel, E. J.; et al: "A Rapid Prototyping and Integrated Design System for Software Development of GN&C Systems," Advances in the Astronautical Sciences, Vol. 86, 1994, p.91.

[12] Aichoun, Jean-Marc: "Synchronous Data Flow Languages for Onboard Software," *News from Prospace*, No. 37, May, 1995, p. 38.

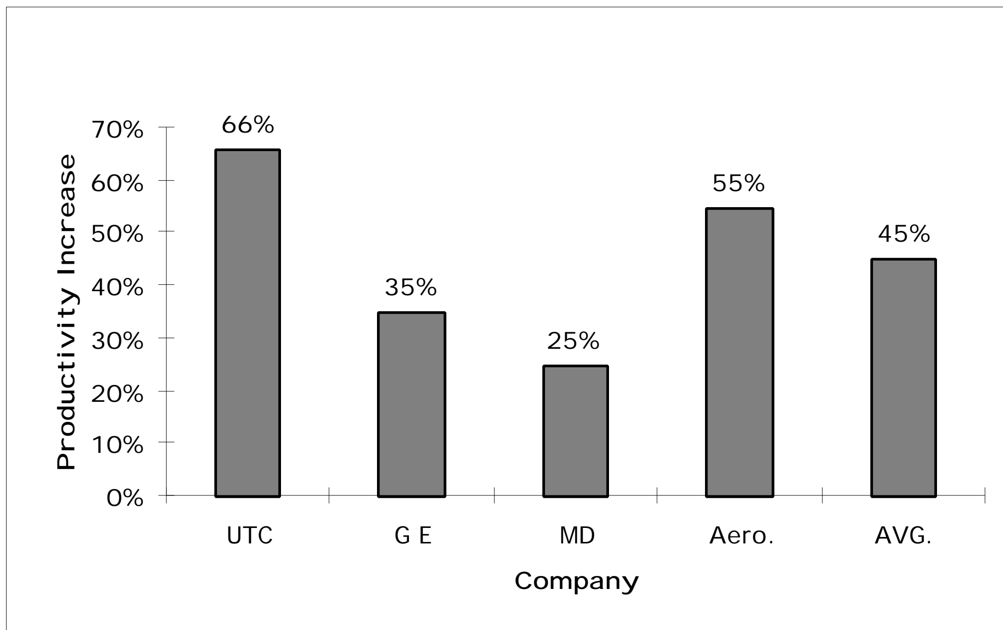


Figure 4.10 Comparison of Productivity Increases

The COnstructive COst MOdel (COCOMO) produced an estimate of 3400 staff-hours to complete the same project with traditional software development approaches. The actual 1830 staff-hours represents a 185% increase in productivity.[13]

As discussed in chapters 1 and 2, the automation based software factory also includes the integration of people and their corporate knowledge. The people access and fine-tune accumulated knowledge as appropriate on new projects. One way to store knowledge is in a central repository of previously created design specifications which can be reused. Lockheed's LEAP environment employs such a scheme with a library of stored templates. LEAP was used to create Sensor Resource Management (SRM) software for a satellite. The SRM code was 70K lines of Ada.

[13] Bordano, Aldo; and Uhde, Jo; et al: Cooperative GN&C Development in a Rapid Prototyping Environment, American Institute of Aeronautics and Astronautics Report No. AIAA-93-4622-CP, 1993, p.889.

> The SRM system was developed in about three months with a two-person level of effort. Our estimate based on this application is that LEAP yielded a productivity gain of 50:1 over the traditional design and coding process. This experiment demonstrated that LEAP effectively produced software for a major application effort with significant productivity gains. [14]

In addition to improving productivity, software factory processes also improve software quality. The United Technologies case study documented an error reduction of 80%, Figure 4.11. Aerospatiale also experienced similar reductions. Table 4.4 shows the growth of on-board software for the Airbus jetliners. Software for the A310 was hand coded and Aerospatiale has been able to reduce coding error by 88% or greater through the use of automatic code generation on the A320 and A340 programs. Automatic code generation accounted for 70% of the Airbus A340 code.

Table 4.4 Encoding Errors in Airbus Software[15]

Aircraft	**A310**	**A320**	**A340**
Flight Software (Megabytes)	4	10	20
Errors per 100 Kilobytes	100	12	10

[14] Baker, James; and Graves, Henson: "Simulation Based Software Development," Proceedings of the 8th Annual International Computer & Applications Conference, November, 1994, p. 123.
[15] Aichoun, Jean-Marc: op cit, p. 37.

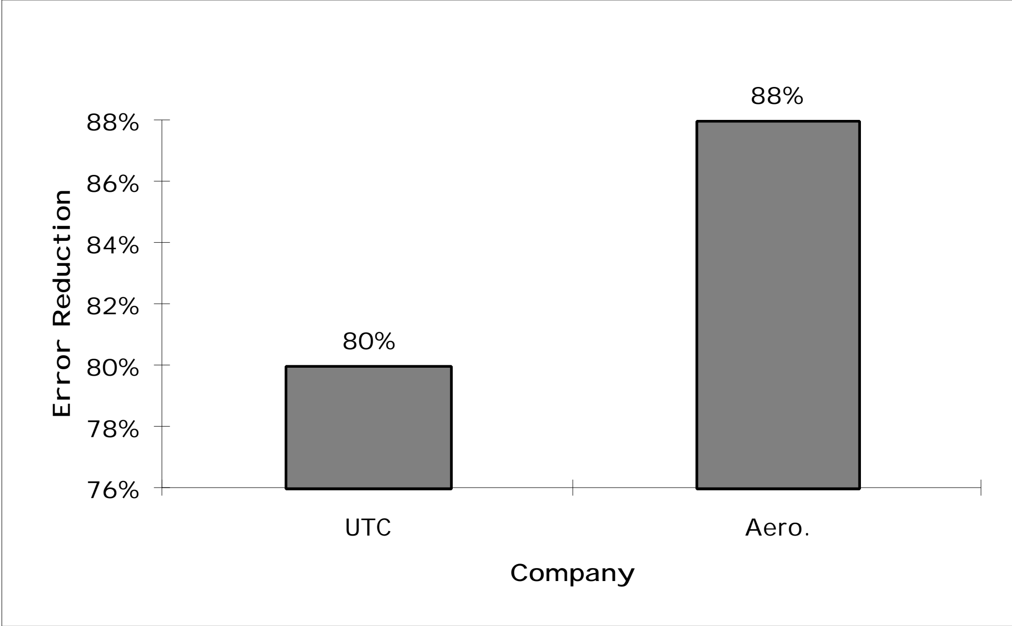


Figure 4.11 Comparison of Reductions to Encoding Errors

The Benefits and Implementation of the Software Factory

There now exists a base of experience in the aerospace industry demonstrating that software factory development and software factory CASE technologies can be used to create safety and mission critical software for aircraft. An approximate productivity increase of 45% can be expected and significantly greater gains are achievable, all while increasing product quality. Additional benefits are also realized with software factory development, Table 4.5.

Table 4.5 Shared Benefits of Software Factory Development

- Increased productivity
- Decreased errors and improved reliability
- Enhanced reuse of designs, simulations and test cases
- Increased discipline and adherence to the development process
- Decreased focus on coding, debugging and unit testing
- Increased focus on system requirements and proper design
- Ability to implement rapid prototyping
- Rapid responses to requirement changes
- Barriers between separate functional roles are broken
- Software is easier to maintain

The following excerpts from published papers and articles testify to these additional benefits.

> By allowing the systems designers to capture the information in a graphical form, and automating the generation of code from reusable templates, faster development times are experienced. Project documentation is enhanced and becomes a by-product of the design process. The code generated from the reusable templates is more reliable than hand generated code, because the templates have been thoroughly tested, and have been used on many applications. Because the reliability of the generated code, unit or modules testing can be minimized, or possibly eliminated in favor of immediate integration testing.[16]
>
> The automatic generation of code not only eliminates the time-consuming manual coding, but also avoids the manual introduction of bugs into the code.[17]

> The key to bringing down total development times is the ability to rapidly prototype the system. The engineers on the MSTI project achieve this by developing and using reliable models of multiple complex custom-built devices at an early stage of the design by using the MATRIXx product family supplied by ISI. If changes are made to the specifications, simulations can run the modified models often within minutes, and with a high degree of confidence.
>
> Another benefit of these tools is the ability of project engineers with very different backgrounds to communicate using a common interface. This allows software engineers and control engineers to discuss system models represented as blocks in MATRIXx graphical programming environment (SystemBuild) on equally familiar terms and devote more time to refining the system as a whole.[18]

---

[16] Dellen, Chester; and Liebner, Greg: “Automated Code Generation from Graphical, Reusable Templates,” Proceedings of the 10th Digital Avionics Systems Conference, October, 1991, p. 299.
[17] Rimvall, C.M.; et al: “An Open Architecture for Automatic Code Generation using the BEACON CACE Environment,” Proceeedings of the IEEE/IFAC Joint Symposium of Computer-Aided Control System Design, March, 1994, p. 315.
[18] Mirab, Hamid: “MATRIXx in MSTI,” *Space*, Jan/Feb, 1995, pp. 3-4.

> A unique feature of RAPIDS is that the designer is involved in the process for the whole design, software development, and validation process. This is consistent with concurrent engineering approaches which have historical track records of substantial efficiency improvement. Since the design teams are smaller and more integrated - human errors are fewer, requirements are more traceable, simulations are more usable, and major errors or design flaws are uncovered earlier in the program mitigating costly changes to designs during system integration and test phases. An integrated design team means that the distinctions between systems engineering, GN&C engineering, and software engineering overlap - a single team member may be called upon to fulfill these functions over the course of a program. However there is one product (i.e. a system level design, implemented in software that meets the system level requirements). The design team takes ownership of the entire process and end product. This differs from a traditional process in which each technology discipline owns only a portion of the final design and for only a certain phase of the program. End-to-end responsibility and ownership tends to be more efficient and promotes a more productive work environment for the designers.[19]

Also, lessons learned from industry experiences were compiled from the research. These lessons form the keys to implementing a software factory, Table 4.6.

[19] Maras, M. G.; and Riel, E. J.; et al: op cit, p.93.

Table 4.6 Software Factory Implementation

- Create a strategic plan
  - Focus on a single product line or specific software domain
  - Establish clear quality and productivity goals for any improvement initiatives
- Implement a standardized software development process
  - Capture fundamentals of engineering practice and management
  - Explicitly define work flows and tasks
- Select CASE technologies that collaborate to support and automate the process
  - Automation of process tasks, documentation and metric collection
  - Support of project management, process control and configuration control
  - Leverage of requirements analysis and design
- Select CASE technologies that span the entire process life cycle and support reusable components
  - Include a capable automatic code generator
  - Emphasize reuse of design specifications over reuse of source code
- Treat CASE technologies as one part of an integrated approach to improvement
  - Provide for management integration, process integration, team integration, and tool integration
  - Capture the learning and knowledge of the organization
  - Capture economies of scope by sharing resources and components across projects
- Invest in research and development
  - Develop CASE technologies in-house if required
  - Accommodate upgrades and evolving CASE technologies
- Provide training and an organizational framework
  - Increase employee skill-sets
  - Avoid distinct functional roles and promote teams
  - Expect and address resistance from all levels
- Maintain a long-term view and commitment while measuring progress

Chapter 5: Conclusions

This report documents research of the software factory as a way to improve software development. The software factory is characterized by the use of CASE technology in a highly integrated environment that automates the life cycle and captures engineering knowledge. Use of the software factory is a "lean" manufacturing practice which improves both productivity and product quality while decreasing the number of people needed to develop an operational flight program. The benefits of a software factory (Chapter 4) suggest several "lean" principles are enabled by the software factory, Table 5.1.

Table 5.1 Lean Principles enabled by the Software Factory

- Assure seamless information flow
- Optimize capability and utilization of people
- Implement integrated product process development
- Ensure process capability and maturation

Observations and Findings

Over the last six years there has been a growing base of experience in the aerospace industry that demonstrates software factory development and software factory technologies can be used to create safety and mission critical software for aircraft. An approximate productivity increase of 45% can be expected and significantly greater gains are achievable, all while increasing product quality. However, the benefits of the software factory have yet to be taken advantage of by the whole industry.

Table 5.2 Observations and Findings on SW Factory Development and Technologies

- Applicability to safety/mission critical flight software has been demonstrated
- Improvements to productivity and quality have been realized
- Currently only used for guidance, navigation and control software domain
- Primary used for demonstration projects, limited use on production programs
- Technology is entering a commercialization phase
- Long term strategic management initiatives needed to create software factory

The use of software factory practices is currently confined to the software domain of guidance, navigation and control (GN&C). GN&C is a domain that has historically represented designs as schematic block diagrams making it the domain for automatic code generation technology to naturally develop. With the exception of the product line of EEC software for production engines by Pratt & Whitney and General Electric, automatic code generation has only been used on demonstration projects or special development programs like the Delta Clipper-Experimental (DC-X) or the Miniature Sensor Technology Integration (MSTI) satellite.

Several companies have demonstrated a long term strategic commitment to software factory primarily through the investment in research and development programs. For example United Technologies, General Electric, and Lockheed all developed their own automatic code generation technologies. The need for R&D programs was driven by the fact that commercial CASE tool vendors primarily supported developers of business and information systems rather than developers of real-time engineering systems. However, a phase of commercialization is beginning as corporations look to third parties to maintain the developed technologies. For example, the Verilog SAO+SAGA environment incorporates technology developed by Aerospatiale. In 1995, Honeywell formed a partnership with

MGA Software of Minneapolis, MN to integrate the DSSA/Honeywell automatic code generator with MGA simulation tools. GE is commercializing the BEACON environment through a similar agreement with Applied Dynamics International (ADI) of Ann Arbor, MI. ADI made BEACON available for purchase in December, 1995. As these products compete with the MATRIXx product family in the commercial market, prices should drop and the rate of development increase.

Areas for Further Study

Areas for further study include expanding the productivity study and developing a deeper understanding of domain coverage. A productivity study should independently verify the reported increases to productivity and identify which phases of the software development life cycle are most affected. The study should also attempt to identify the maximum achievable productivity gains.

Also, coverage of software domains by automatic code generation technologies must be addressed. Flight software for the DC-X and Airbus A340 was developed using automatic code generation, but for each vehicle this amounted to only 70% of the total software. The limitations of current automatic code generation technologies need to be understood. However, much of the capabilities of current code generators should be applicable to other domains. General Electric realizes over 80% coverage of a typical control system with their automatic code generation tool (Dellen, 1991). Only 35-30% of the control system coverage is provided by signal-flow block diagrams. The remaining coverage is provided by control-flow diagrams. Control-flow diagrams are traditional flow charts that capture a program’s control structure and processing and can be applicable to all software domains. Table 5.3 shows the distribution of the F-22 operational flight program by software domain. The vast majority is general avionics software. Therefore, automatic code generation technologies must be shown to be applicable to these domains before

software factories can be established for all product lines and productivity gains can achieved across the defense aircraft industry.

Table 5.3 Distribution of F-22 Software Domain[20]

• OFP	100%				
	1. Vehicle Management	4.1%			
	2. Utilities and Subsystems	7.1%			
	3. Avionics	88.8%			
			3.1	Comm./Nav./Identification	26.6%
			3.2	Electronic Warfare	14.4%
			3.3	Mission Software	13.5%
			3.4	Radar	12.4%
			3.5	Controls and Displays	9.4%
			3.6	Core	8.4%
			3.7	Stores Management System	2.1%
			3.8	Inertial Reference System	1.8%

[20] "F-22 Software on Track with Standard Processes," *Aviation Week and Space Technology*, Vol. 143, No. 4, July 24, 1995, p. 53.

Chapter 6: Bibliography

Aichoun, Jean-Marc: "Synchronous Data Flow Languages for Onboard Software," *News from Prospace*, No. 37, May, 1995.

Babel, Philip S.: Software Development Integrity Program, ASD/EN Report, Wright-Patterson AFB.

Baker, James; and Graves, Henson: "Simulation Based Software Development," Proceedings of the 8th Annual International Computer Software & Applications Conference, November, 1994.

Bell, Rodney; and Sharon, David: "Tools to Engineer New Technologies into Applications," *IEEE Software*, March, 1995.

Bell, Rodney; and Sharon, David: "Tools That Bind: Creating Integrated Environments," *IEEE Software*, March, 1995.

Bennett, Keith (ed.): Software Engineering Environments: Research and Practice, Ellis Horwood Limited, 1989.

Bordano, Aldo; and Uhde, Jo; et al: Cooperative GN&C Development in a Rapid Prototyping Environment, American Institute of Aeronautics and Astronautics Report No. AIAA-93-4622-CP, 1993.

Chmura, Alan; and Crockett, Henry: "What's the Proper Role for CASE Tools?" *IEEE Software*, March, 1995.

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

Defense Acquisition University: Intermediate Systems Acquisition Course, Volume IV, January, 1996.

Dellen, Chester; and Liebner, Greg: "Automated Code Generation from Graphical, Reusable Templates," Proceedings of the 10th Digital Avionics Systems Conference, October, 1991.

Endres, A.; and Weber, H. (eds.): Software Development Environments and CASE Technology, Springer-Verlag, 1991.

Englehart, Matt; et al: ControlH Programmers Manual, Version 11.8, Honeywell Technology Center, March, 1995.

"F-22 Software on Track with Standard Processes," *Aviation Week and Space Technology*, Vol. 143, No. 4, July 24, 1995.

Fisher, Alan S.: CASE: Using Software Development Tools, Wiley, 1988.

Graves, Henson: "Lockheed Environment for Automatic Programming," *IEEE Expert*, December, 1992.

Honeywell: DoME Users Manual, Version 3.20, Honeywell Technology Center, October, 1995.

Hou, Alex C.: Toward Lean Hardware/Software System Development: Evaluation of Selected Complex Electronic System Development Methodologies, MIT Lean Aircraft Initiative Report LEAN 95-01, February, 1995.

Johnson, D. M.: “The Systems Engineer and the Software Crisis,” *Software Engineering Notes*, Vol. 21, No. 2, March, 1996.

Jones, Denise; and Turkovich, John; et al: “Automated Real-Time Software Development,” Proceedings of the 3rd National Technology Transfer Conference & Exposition, NASA Conference Publication 3189, Volume 2, 1993.

Lavi, Jonah Z.: “Development of a Method Driven CAS^2E Tool,” Software Development Environments and CASE Technology, Springer-Verlag, 1991.

Macala, Randall R.; et al: “Managing Domain-Specific, Product-Line Development,” *IEEE Software*, May, 1996.

Maras, M. G.; and Riel, E. J.; et al: “A Rapid Prototyping and Integrated Design System for Software Development of GN&C Systems,” Advances in the Astronautical Sciences, Vol. 86, 1994.

Mazzucchelli, Lou: “You’re in the Software Army Now,” *Electronic Business*, April 27, 1992.

Mettala, Erik; and Graham, Marc: “The Domain Specific Software Architecture Program,” Proceedings of the Software Technology Conference, DARPA, 1992.

Mirab, Hamid: “MATRIXx in MSTI,” *Space*, Jan/Feb, 1995.

Newport, John R.: Avionic Systems Design, CRC Press, 1994.

National Institute of Standards and Technology (NIST)/ECMA: Reference Model for Frameworks of Software Engineering Environments, NIST Special Publication 500-211, Edition 3, August, 1993.

O’Connor, James; et al: “Reuse in Command-and-Control Systems,” *IEEE Software*, September, 1994.

Ogata, Lori; and Jensen, Paul: “Automatic Programming for Rapid Development of Tracking Systems,” Proceedings of the IEEE/AIAA 10th Digital Avionics Systems Conference, October, 1991.

Over, James W.: “Software Process Asset Library,” Proceedings of the Software Technology Conference, DARPA, 1992.

Paulk, Mark C.; et al: The Capability Maturity Model For Software, Version 1.1, Software Engineering Institute Report No. CMU/SEI-93-TR-24, Carnegie Mellon University.

Reil, Ed J.; et al: "A Rapid Prototyping and Integrated Design System for Software Development," Proceedings of the AIAA 9th Computing and Aerospace Conference, October, 1993.

Rimvall, C. Magnus; et al: "An Open Architecture for Automatic Code Generation using the BEACON CACE Environment," Proceeedings of the IEEE/IFAC Joint Symposium of Computer-Aided Control System Design, March, 1994.

Rimvall, C.; and Radecki, M.; et al: "Automatic Generation of Real-Time Code using the BEACON CAE Environment," Proceedings of the 12th Triennial World Congress of the International Federation of Automatic Control, Vol. 2, 1993.

Sage, Andrew P.; and Palmer, James D.: Software Systems Engineering, Wiley, 1990.

Schefstrom, D. (ed.): Tool Integration: Environments and Frameworks, Wiley, 1993.

Somerville, Ian: Software Engineering, Addison-Wesley Publishing Co., 1996.

Spang, H. A.; et al: "The BEACON Block-Diagram Environment," Proceedings of the 12th Triennial World Congress of the International Federation of Automatic Control, Vol. 2, 1993.

Turkovich, John: "Automated Code Generation for Application Engineers," Proceedings of the 9th Digital Avionics Systems Conference, October, 1990.

Uhde, Jo; et al: Library Reuse in a Rapid Development Environment, American Institute of Aeronautics and Astronautics Report No.AIAA-95-1015-CP, 1995.

Vestal, Steve: MetaH Programmers Manual, Version 1.01, Honeywell Technology Center, March, 1995.

Walker, Carrie; and Turkovich, John: "Computer-Aided Software Engineering: An Approach to Real-Time Software Development," Proceedings of the 7th AIAA Computers in Aerospace Conference, October, 1989.