Building Software Factories in the Aerospace Industry

by

Jose K. Menendez

B.S. Aerospace Engineering
University of Michigan, 1988

Submitted to the Department of Aeronautics and Astronautics
in Partial Fulfillment of the Requirements for the Degrees of

MASTER OF SCIENCE in TECHNOLOGY AND POLICY

and

MASTER OF SCIENCE in AERONAUTICS AND ASTRONAUTICS

at the
Massachusetts Institute of Technology

February 1997

Signature of Author: ____________________
January 17, 1997

Certified by: ____________________
John Deyst
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by: ____________________
Professor Richard de Neufville
Chairman, Technology and Policy Program

Accepted by: ____________________
Jaime Peraire
Professor of Aeronautics and Astronautics
Chair, Graduate Office

Aero

THIS PAGE INTENTIONALLY BLANK

Building Software Factories in the Aerospace Industry

by Jose K. Menendez

Submitted to the Department of Aeronautics and Astronautics in Partial Fulfillment of the Requirements for the Degrees of Master of Science in Technology and Policy and Master of Science in Aeronautics and Astronautics

ABSTRACT

The defense aerospace industry is currently in a phase of shrinking procurement budgets brought on by the end of the Cold War and pressures to reduce the national deficit and balance the Federal budget. Consequently, the Department of Defense has shifted its product development emphasis from system performance to system affordability. Simultaneously, software has become increasingly important for implementing functionality in new systems and sometimes dominates total product development costs. The challenge for industry is to implement new processes and technologies that will allow the reliable, repeatable development of high quality software at reduced cost. One emerging practice capable of meeting this challenge is the software factory.

A software factory is an industrialized approach to software development which exploits technologies such as automatic code generation tools to secure cycle time reductions and quality improvements. Software factories also generate longer term savings by producing reusable assets suitable for future applications, allowing an organization to achieve economies of scope.

This thesis presents an overview of numerous technology development efforts in the aerospace industry which facilitate software factory development. A detailed case study of United Technologies is also presented. Productivity and quality metrics were collected to support a comparative analysis of the development of electronic engine control system software for the PW4000 Current and PW4000 Growth aircraft engines, which employed traditional and factory development methods respectively. Data reported by additional companies experimenting with software factory development has been compiled indicating that an immediate productivity improvement of 45% is consistent for the industry.

Despite clear successes, software factory processes are not in widespread use across the industry and experiences have been focused on a rather narrow range of applications for guidance, navigation, and control systems. Software factory processes have also been primarily applied in demonstration projects. Yet there are no insurmountable reasons why the software factory could not be applied to other software domains and production programs. This thesis closes with a discussion of challenges facing the widespread use of the software factory and the role policy plays in promoting its adoption.

KEYWORDS

automatic code generation, computer-aided software engineering (CASE), domain engineering, integrated environments, product-line development, software development productivity, software factory, software reuse, software quality

Thesis Supervisor: John Deyst
Title: Professor of Aeronautics and Astronautics

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THIS PAGE INTENTIONALLY BLANK

Chapter 1: Introduction

The United States military has become dependent on all types of computer systems to accomplish its mission as shown in Table 1.1. In 1995 alone, the Department of Defense spent nearly 43 billion dollars to purchase computer systems (STSC, 1994). Only 7 billion of this expenditure purchased hardware, with the remaining 35 billion dollars going towards the development and maintenance of software. Software has become the major factor affecting the cost of computer systems and the cost growth in software has also outpaced the cost growth in hardware. Figure 1.1 shows the costs for all Department of Defense computer systems for a period from 1985 to 1995 (STSC, 1994). Hardware costs increased by a factor of 2 while software costs increased by a factor of 3.

One of the many types of DoD software contributing to this cost growth is the real-time software embedded in weapon systems such as aircraft. Real-time embedded software has become increasingly important for implementing avionic systems functionality in all types of aircraft. Avionic systems in modern defense aircraft are highly sophisticated and complex. They are composed of multiple subsystems (navigation, radar, flight control, engine control, warfare systems, etc.) distributed over multiple processors throughout the aircraft. Software is embedded into each subsystem and is used to provide for overall system integration, making it critical to both the mission and safety of the aircraft. The increasing dependence of aircraft systems on embedded software is evident in Figure 1.2 (Babel) which illustrates an exponential growth of on-board memory size.

The growth in development costs and the amount of software to be developed have combined to form a major challenge for the aerospace industry: to develop high quality software in a reliable, repeatable manner while improving productivity. New processes and technologies are constantly being implemented to meet this challenge. One emerging

Table 1.1 Types of DoD Computer Systems

Weapon Systems	Management Information Systems
Embedded Systems	Information System Resources
Command and Control Systems	Automated Information Systems
Intelligence and Cryptological Systems	Information Resource Management
All other weapon	All other non-weapon

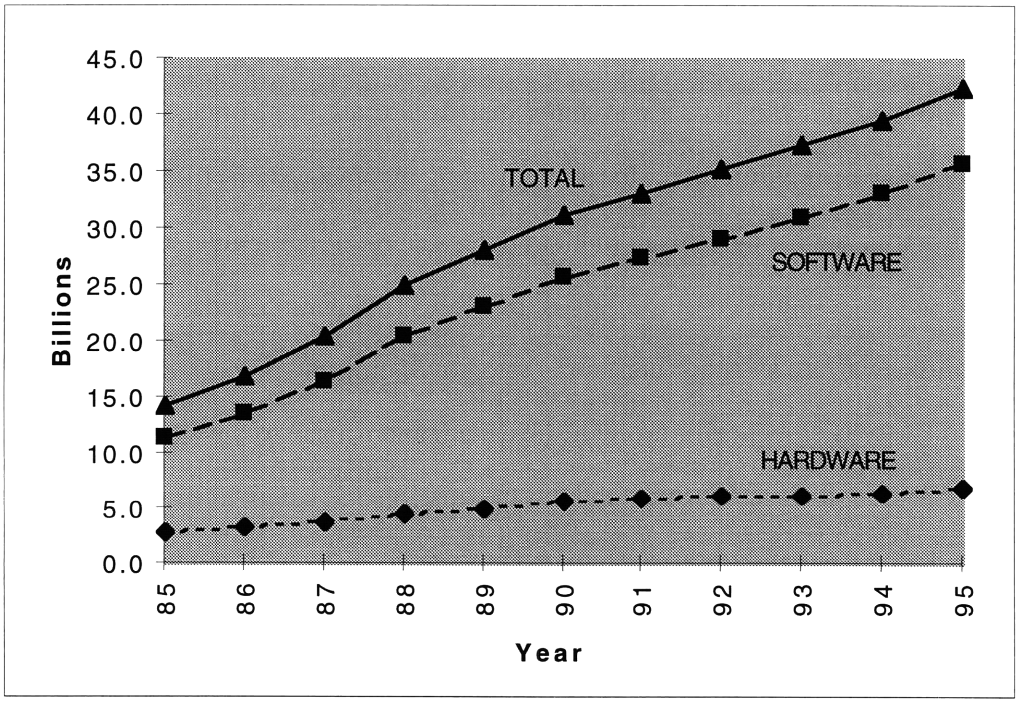


Figure 1.1 Annual Costs of DoD Computer Systems (all types)

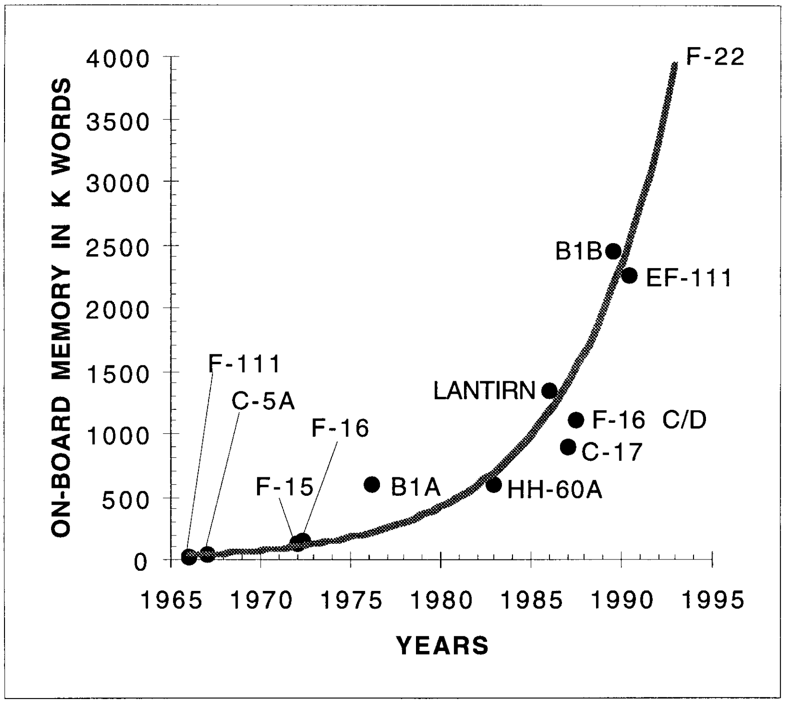


Figure 1.2 Growth of Aircraft Software

practice, attempting to change the nature of software development from a craft to mass production, is the software factory.

Many organizations within the aerospace industry have begun to build software factories. A software factory is an industrialized approach to software development with the goal of using systematic reuse of design specifications to achieve economies of scope. This thesis takes an in depth look at the software factory and the technologies which support software factory development in an attempt to determine the impact they are having on the industry.

Outline of Thesis

Chapter 2 provides the context for the thesis by presenting an overview of the development of software for Department of Defense weapon systems. The chapter also introduces the concept of a software factory in greater detail. Added emphasis is given to supporting practices such as the integration of computer-aided software engineering (CASE) technologies into a software factory environment and the use of automatic code generation.

Chapter 3 presents an overview of numerous technology development efforts in the aerospace industry. A description of integrated environments and CASE technologies developed by aerospace corporations, commercial tool vendors, and government agencies is included. Common characteristics of the technologies resulting from these parallel efforts are compiled.

Chapter 4 discusses experiences within the aerospace industry with using these technologies and other software factory practices. A case study of software factory development at United Technologies forms the bulk of this chapter. Software developed for similar products and created by different development practices (traditional and factory) forms the basis of a comparative analysis. This analysis quantifies the impacts of factory practices on the productivity and quality of the development process. Data from additional companies has also been compiled in an effort to determine if the experiences of United Technologies are consistent with others in industry and to identify the expected benefits of implementing a software factory.

Chapter 5 begins with a summary of observations and findings. A diffusion model illustrating the widespread industry adoption of the software factory for production programs is presented and followed by a discussion of issues requiring resolution. The thesis concludes with policy recommendations and the identification of areas for further research.

The Lean Aircraft Initiative and Acknowledgments

The research of this thesis has been supported by the Lean Aircraft Initiative (LAI). Sponsored by the U.S. Air Force and private corporations, the LAI is a research program jointly managed by the USAF Aeronautical Systems Center and the Massachusetts Institute of Technology. The LAI mission is to "define and help implement roadmaps for fundamental change in both industry and government operations, based on best "lean" practices, resulting in: greater affordability of systems, increased efficiency, higher quality, enhanced technological superiority, and a stronger U.S. defense aircraft industrial base."

The LAI takes a broad view of the defense aircraft industry, encompassing all sectors of defense aircraft production, including airframe integrators and major supplier groups, such as producers of engines, avionics and electronic systems, and other equipment suppliers. The program has undertaken research in five major areas: Product Development, Factory Operations, Supplier Systems and Relationships, Organization and Human Resources, and Policy and External Environment.

There are numerous members of the Product Development Focus Group whom I wish to recognize for their contribution to this thesis. I wish to thank John Deyst, Charlie Boppe, Dave Hoult, John Miller, and all the other MIT members of LAI who provided encouragement and guidance. Also, I wish to thank the many from industry, including Al Klein (Hamilton Standard), Jim Krodel (Pratt & Whitney), J.J. Liu (Hughes), and Marvin Ross (Hughes), who provided data and critical feedback. Their time and cooperation are greatly appreciated. Although supported by LAI, the opinions expressed in this thesis are strictly those of the author and not those of LAI sponsors.

THIS PAGE INTENTIONALLY BLANK

Chapter 2: Defense Software Development and the Software Factory

A major challenge for the defense aerospace industry is to develop high quality software in a reliable, repeatable manner while improving productivity. New processes and technologies are constantly being developed to meet this challenge. These developments have led to the emergence of the software factory. This chapter establishes the context for the thesis by presenting an overview of the development of software for Department of Defense weapon systems and providing a detailed introduction to the software factory and its supporting practices.

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. Standards are used to guide the activities of defense contractors by defining processes, methods, and engineering practices. Although recently superseded for new system acquisitions by 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 2.1 illustrates the life cycle and structured approach specified in DOD-STD-2167a. The process starts with users and systems analysts defining system level requirements for a hardware/software system, after which separate parallel hardware

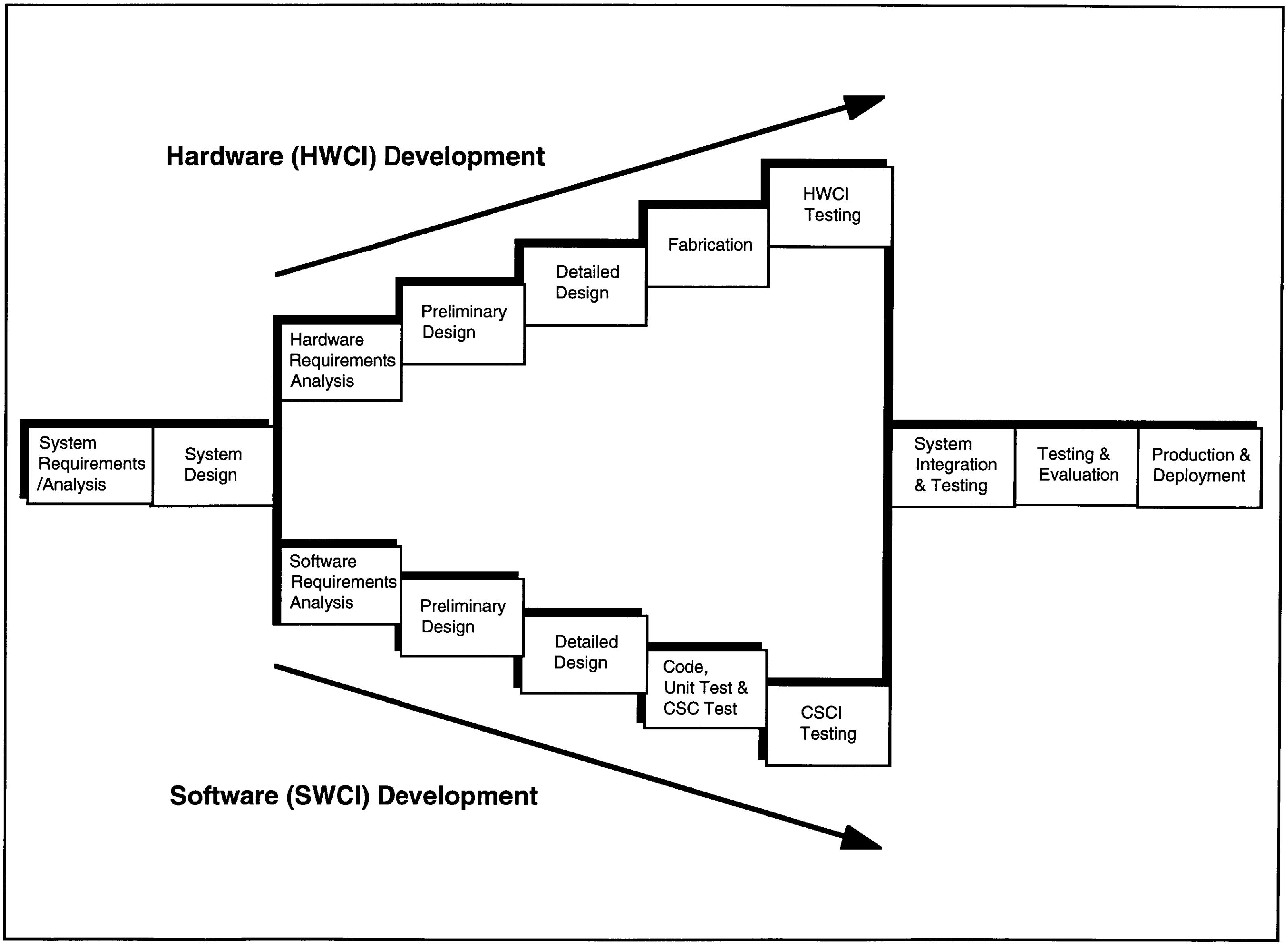

Hardware (HWCI) Development
System Requirements /Analysis
System Design
Hardware Requirements Analysis
Preliminary Design
Detailed Design
Fabrication
HWCI Testing
Software Requirements Analysis
Preliminary Design
Detailed Design
Code, Unit Test & CSC Test
CSCI Testing
System Integration & Testing
Testing & Evaluation
Production & Deployment
Software (SWCI) Development

Figure 2.1 DOD-STD-2167a System Development Life Cycle

and software development efforts begin. Hardware and software requirements analyses are conducted to determine partitioning, with traceability back to all system requirements. Requirements analysis is followed by the design phase, broken into preliminary and detailed design. The preliminary design phase defines the software architecture and allocates software functional requirements to program modules. The detailed design phase defines the internal specifics of each module and intermodular interfaces. The designs require implementation and testing. Implementation includes coding, debugging and re-coding (if necessary) while testing 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.

The DOD-STD-2167a process reflects a classic waterfall life cycle which implies that each phase is completed only 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 evolved from the waterfall model 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

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

listed order. MIL-STD-498 is consistent with the acquisition reform initiatives of Defense Secretary William Perry who stated standards should be viewed as tools for guidance instead of strict requirements.

Regardless of which software development process an organization selects, the organization must have an effective 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 (Paulk, 1993). The Capability Maturity Model (CMM) includes five maturity levels which describe a path from ad hoc, chaotic processes to mature, disciplined software processes.

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.[1]

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 issues related to the organization's software development process, 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

[1] 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, 1993, Chapter 2.

monitor the state of the software process used on an existing software effort. Software capability evaluations are typically conducted by independent evaluation teams.

Problems Affecting Software Development

Difficulties persist with weapon systems software development despite Department of Defense efforts to positively influence software development management by defense contractors. Traditional software development is characterized by a hand-crafted approach with different specialists responsible for separate phases (i.e. requirements analysis, design, implementation, test and maintenance). The numerous problems with this approach, Table 2.2 (Hopkins, 1993), result in the introduction of errors, Figure 2.2 (Babel). The labor intensive nature of the craft approach also affects quality, which is difficult to maintain and control when large development teams are required, and often results in diseconomies of scale. It is common for average productivity to decrease as the number of developers on a project increases. Frequently, the end result is cost and schedule overruns as all detected errors must be removed before high quality software can be fielded.

People, process, and technology are the major determinants of software quality and productivity, Figure 2.3 (Over, 1992). Together these three form the leverage points which must be addressed in order to attack the problems of traditional software development and reduce costs. One such holistic approach to software development is the software factory.

Table 2.2 Major Problems of Traditional Software Development

- Inadequate requirements analysis and review at major design reviews
- Inadequate software development planning
- Underestimate of time required for software analysis/design/coding
- Underestimate of time required for software testing and debugging
- Lack of adequate metrics to measure software development progress
- Lack of standardized software development techniques
- Immaturity of software development process
- Inappropriate life cycle model used
- Inadequate development facilities/tools
- Excessive memory/throughput requirements
- Real-time performance shortfalls
- Insufficient documentation

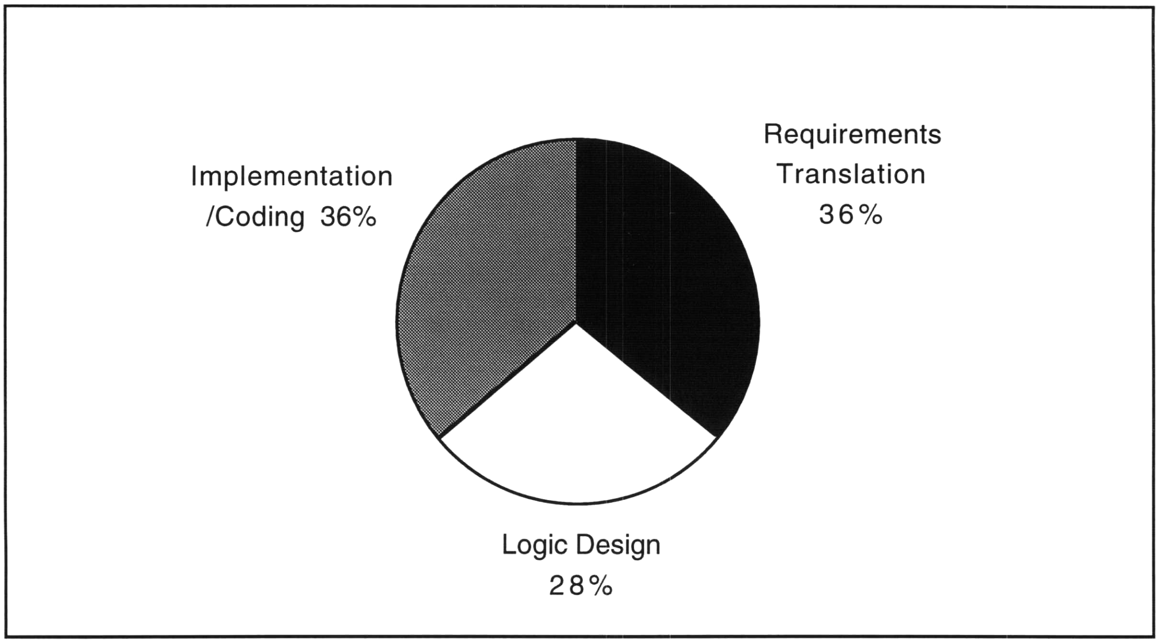


Figure 2.2 Software Error Types During Weapon System Development

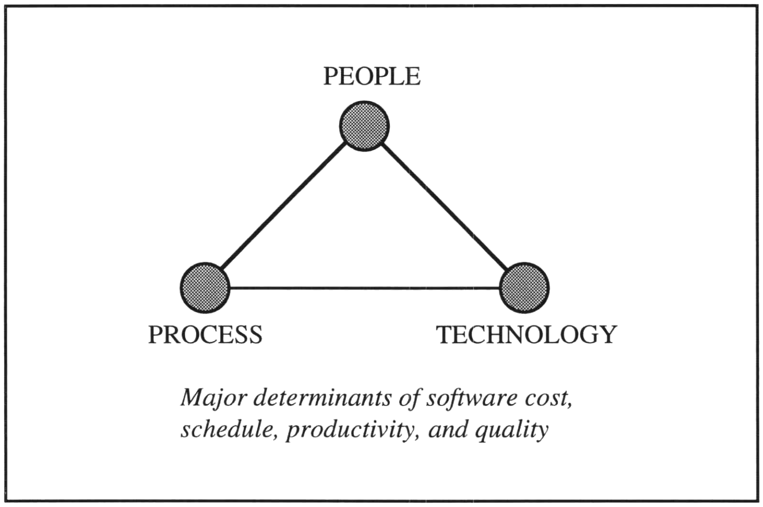

PEOPLE
PROCESS
TECHNOLOGY
Major determinants of software cost,
schedule, productivity, and quality

Figure 2.3 The Three Leverage Points of Software Development

The Software Factory

A software factory is an industrialized approach to software development with the goal of using systematic reuse to achieve economies of scope, Table 2.3. Economies of scope can be achieved within a software organization by reusing software assets across multiple projects. Software factories are characterized by the extensive use of highly integrated computer-aided software engineering (CASE) technologies to form environments supporting process automation, with electronic repositories to store and retrieve software assets. By using standardized automation based development processes, software factories are able to leverage the efforts of personnel toward the early phases of development and implement product teams. Additionally, commitments to continuous improvement and strategic management insures that the factory incorporates the latest development techniques and technologies. Strategic management also involves establishing productivity goals for the software factory and actively addressing any organizational resistance to factory implementation.

Table 2.3 Elements and Objectives of the Software Factory

ELEMENTS	OBJECTIVES
• Product-Line Application Development • Domain Engineering • Integrated Environments and Electronic Repositories • Integrated Product Teams and Training • Continuous Improvement • Strategic Management with Productivity Goals	• Systematic Reuse • Economies of Scope

Product-Line Application Development and Domain Engineering

Software factories achieve reuse across projects by focusing on a single product line aided by domain engineering. A product-line is a family of systems that possess similar functionalities and share a common architecture. A software-intensive product-line can be either narrowly defined, such as radar systems or GPS-based navigation systems, or more broadly defined, such as fighter aircraft or airlifter. Product-line application development creates new software systems by incrementally building or extending from a common architecture through the assembly of reusable assets.

To support product-line application development, software assets (specifications, code, etc.) must be designed with reuse in mind, which is made possible by domain engineering. A domain is a distinct functional area within an underlying set of common requirements and capabilities. Control systems is an example of a software domain. There may be more than one domain within a given product-line and a given domain may support multiple product-lines.

> Domain engineering involves those engineering activities necessary to implement a planned, systematic, product-line based reuse strategy. This strategy often includes: economic analysis, domain scoping and domain analysis, domain specific architecture design, reusable component development, domain-specific software tool development, and reuse process definition and documentation. ... Domain engineering defines which processes, methods, tools, and procedures to use to engineer the software product.[2]

[2] Software Technology Support Center (STSC): Guidlines for Successful Acquisition and Management of Software Intensive Systems, Version 1.1, U.S. Air Force, February, 1994, p. 4-6.

Computer-Aided Software Engineering Technology

Computer-Aided Software Engineering (CASE) is the use of support tools to automate much of the software development process. Automation of the implementation and testing phases can be used to reduce the overall labor requirement, significantly reduce errors introduced during coding, and provide leverage toward the front-end stages of the process. Additionally, providing automated support for analysis and design can eliminate many of the errors that occur during the requirements stage of development while improving the reuse of software components. Jones and Turkovich provide further emphasis on the benefits of automation 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.[3]

There are many uses for CASE technology. CASE technology can be used to enforce process control, ensuring that the process is disciplined and repeatable, and to automate source code generation, eliminating manual errors and reducing the need for debugging. Other labor-intensive tasks can also be automated, allowing software development by small teams while leading to a reduction in cycle time and a shift of focus to requirements analysis rather than implementation. CASE technology can also be used to

[3] 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.

collect metric data, facilitating process improvement and schedule and cost estimation. However, before CASE technology can be introduced to improve both the quality and productivity of a software development process, the process must be clearly defined and should reflect the fundamentals of engineering practice. Furthermore, to fully benefit from all the capabilities of CASE technology, extensive integration into a cohesive software factory environment is required.

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 holistic, 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. A software factory requires an integrated environment to support and facilitate its practices.

Keith Bennett compiled a collection of research papers in his book on software engineering environments (Bennett, 1989). In the first part of Bennett's book, dedicated to Software Factories, Christer Fernstrom's chapter reemphasizes the need for a holistic approach to integration while discussing 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.[4]

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 - process, team, management, and tool (Bell and Sharon, 1995).

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

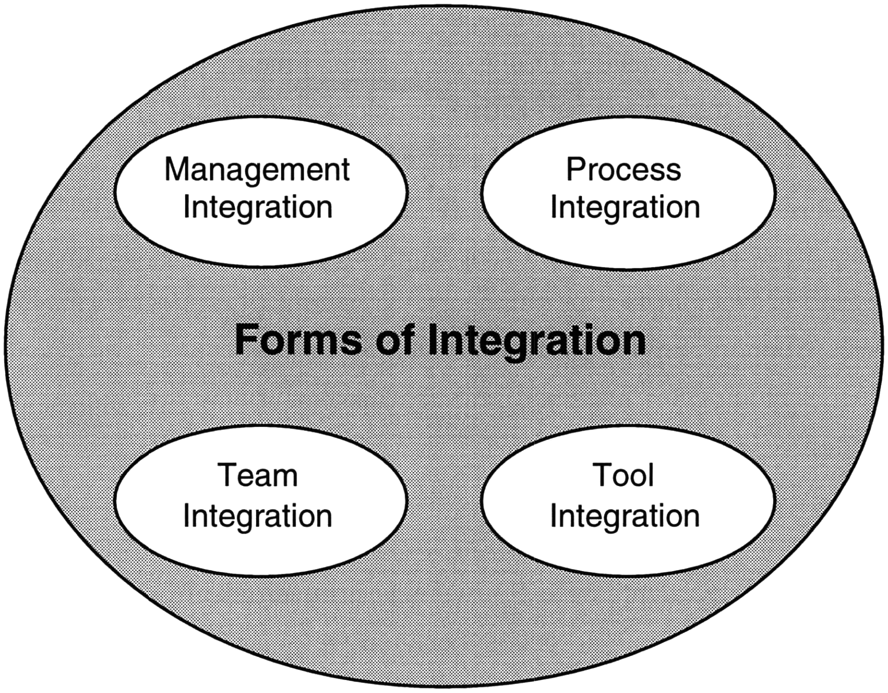


Figure 2.4 Forms of CASE Integration

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 also 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.

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.

Tool integration refers to using CASE tools that share user-interfaces, data and functionality. A detailed discussion of tool integration is presented in the next section of this thesis.

The four forms of integration capture the essential capabilities of an integrated CASE environment: 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 are dependent upon how the tools are developed. Level 1 encompasses tools developed individually. Such tools are likely to have common user interfaces and data import/export formats but are unable to share a single repository. Level 2 encompasses tools developed as a suite. They are tightly integrated and optimized amongst themselves but their integration

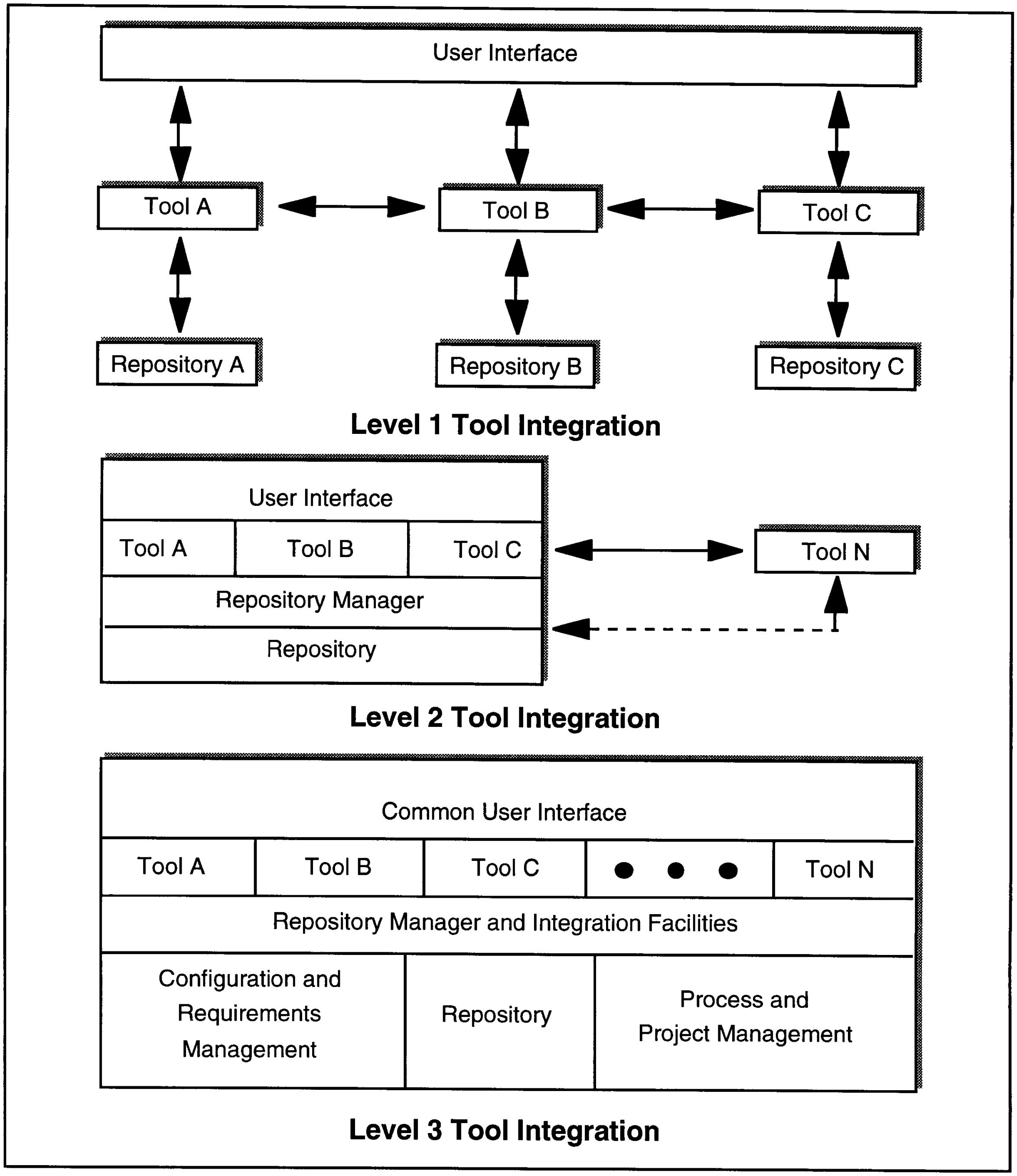


Figure 2.5 **Levels of CASE Tool Integration**

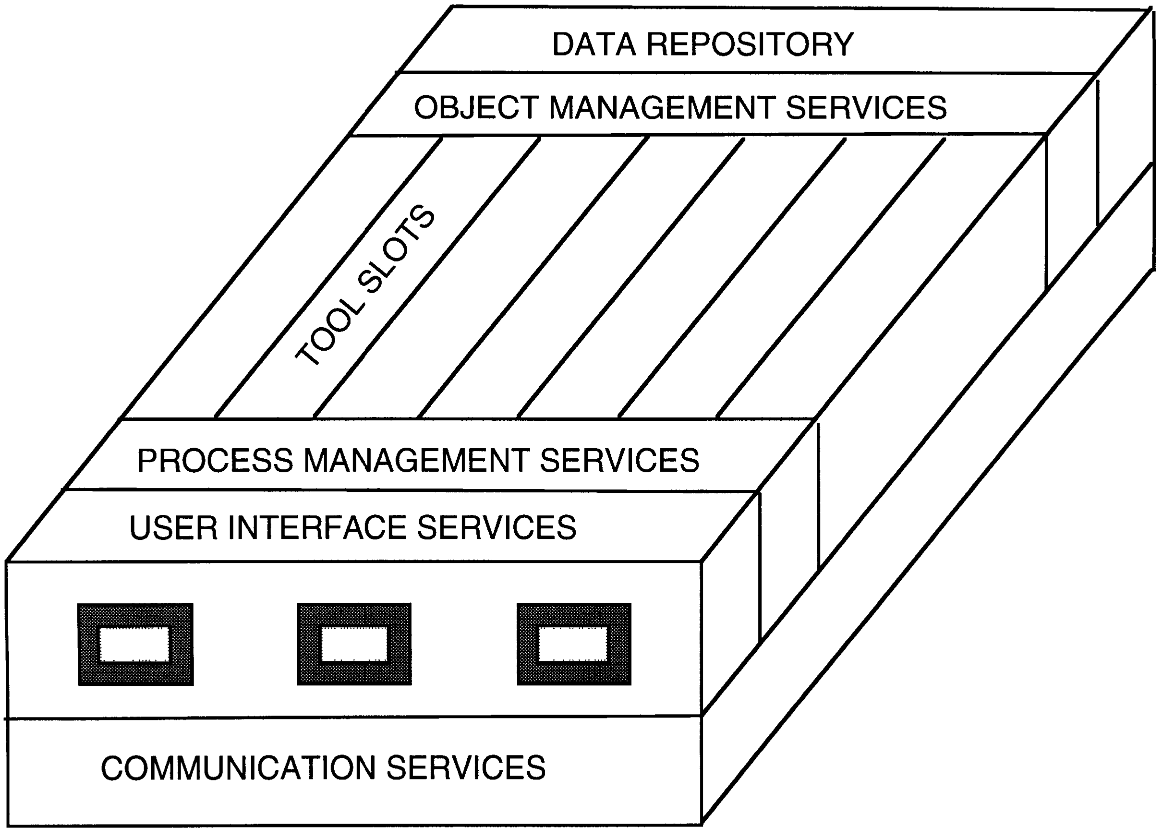


Figure 2.6 NIST/ECMA Reference Model

with other tools remain at a Level 1 capability. Level 3 encompasses tools developed to meet formal interoperability standards such as accredited ANSI standards. These tools may have been developed individually but together they are capable of forming a highly integrated environment.

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

Software factories involve systematic reuse and the use of automation technologies to leverage efforts toward the early phases of software development. Figure 2.7 shows the evolution of CASE technology leading to automatic code generation (Fisher, 1988). Automatic code generation is an emerging CASE technology that aids software factories in both systematic reuse and leverage by allowing a change in focus from the reuse of code to the reuse of design specifications, which is potentially more cost-effective.

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 taking advantage of 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, as shown in Figure 2.2. 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.

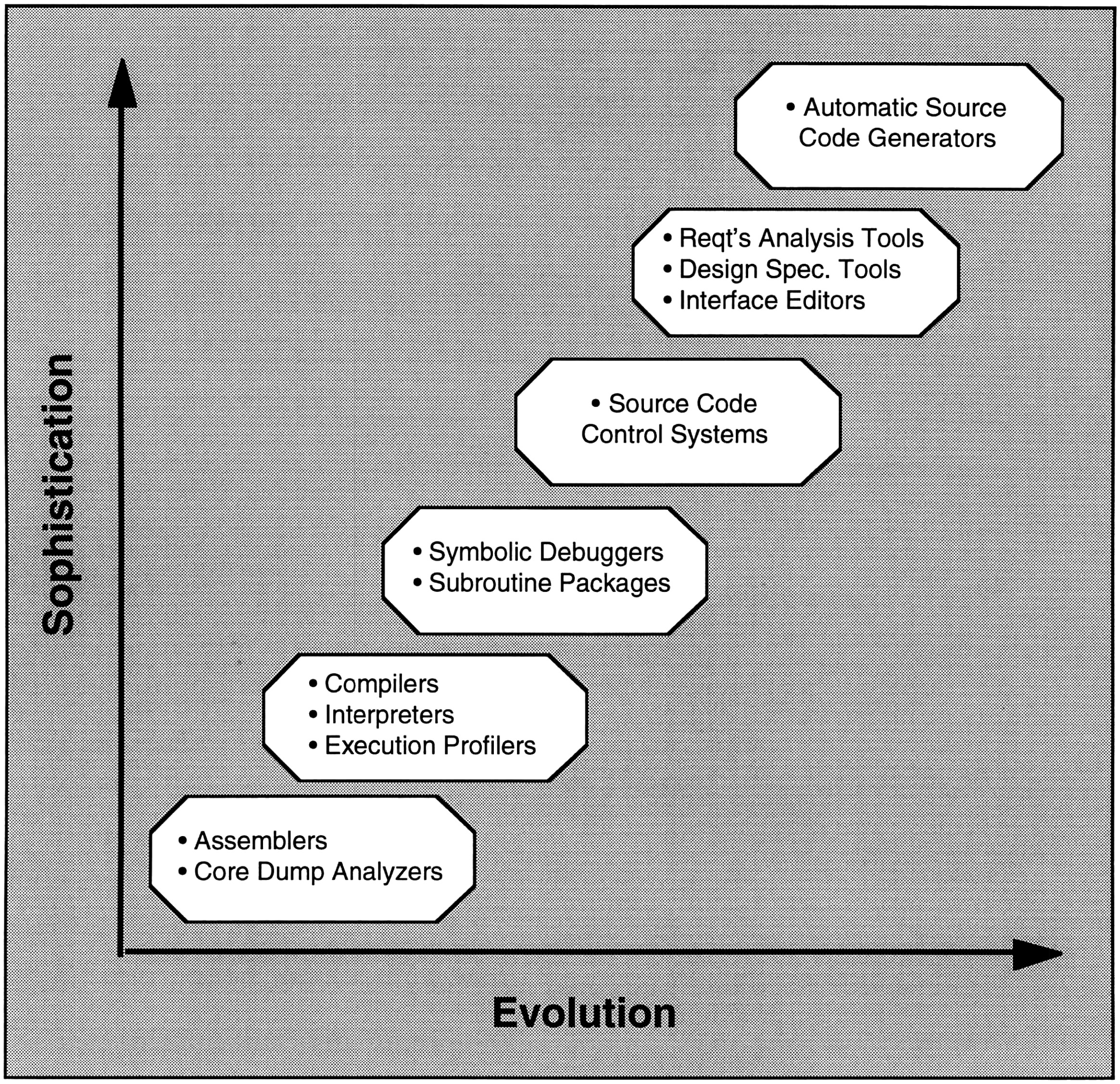


Figure 2.7 Evolution of CASE Technology

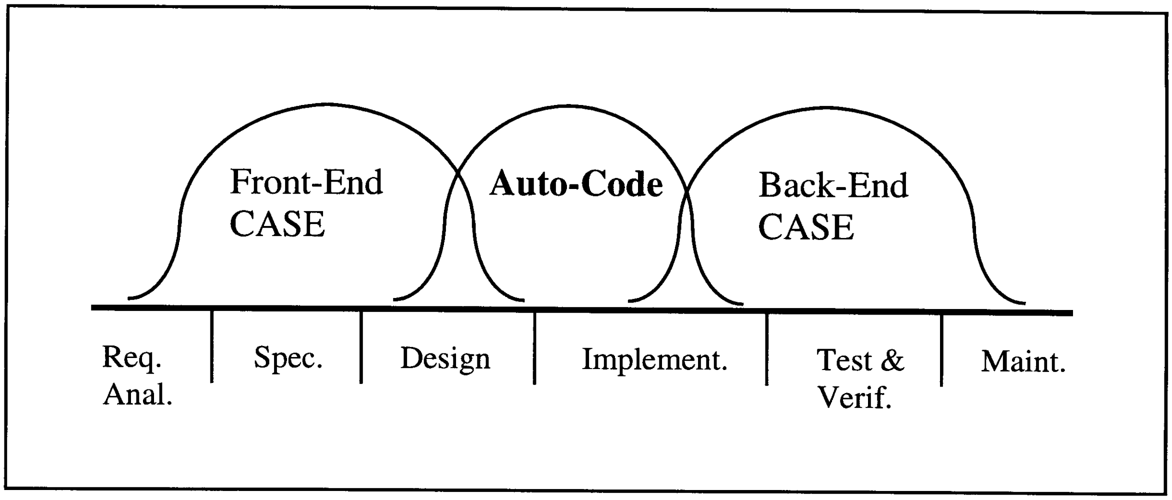


Figure 2.8 Automatic Code Generation bridges the gap in CASE Technology

> 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 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.[5]

Fisher's assessment still holds today. 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

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

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.[6]

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 build a software factory. 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.

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

> 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 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."[7]

When implementing a software factory, management must assume a long-term view and link 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 strategic initiatives. Initiatives include the definition of the product-line, the analysis and standardization of the software development

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

process, investment in integrated CASE technologies, the active promotion of reuse, and the mitigation of resistance.

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 themselves may be specific to a particular domain of application or product-line. And these tools must be flexible enough to support a common process and integrated environment for use across multiple projects.

Training is an additional factor which must be implemented as an integral part of software factory development. Training must include teaching the common process and tool usage to make the development of quality software repeatable and predictable. 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 tendency of automation to reduce skills. But it also breaks barriers of the distinct functional roles between systems and software engineering. Expanding the focus of training beyond tool usage is a strategy for leveraging the front-end of the development process. This allows the linking of CASE technology with organizational changes such as developing software with small integrated product teams in a product-line approach.

Strategic management also involves attention to reuse. Reuse can also be met with resistance and must be actively planned for and encouraged before it is likely to become systematic. Systematic reuse is the highest level of a scale also including ad hoc and opportunistic reuse.

> Ad hoc reuse is a programmer's or workgroup's sporadic use of previously developed code. There is no defined reuse process for identifying and exploiting reuse opportunities, and success relies on the individual's initiative. Opportunistic reuse is characterized by identifying reusable assets

and applying them in a development effort where the software developer identifies and locates the applicable reuse components. This type of reuse is generally project-specific where a general reuse requirement is most likely defined in the project plan. Systematic reuse is reuse planned over a family of product developments with a robust, flexible architecture, where families of reusable assets are developed and maintained to provide large-scale reuse. With this type of reuse, new opportunities for reuse, based on anticipated customer needs and technology innovation, are created.[8]

[8] Department of Defense: DoD Software Reuse Initiative: Strategic Plan, Document No. PD436.6, December 14, 1995, Section 2.1.

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.

Technology development has been underway at various places including aerospace corporations, commercial tool vendors, and government agencies, Table 3.1. 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. This chapter closes by summarizing the common characteristics shared by the developed technologies.

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.	MATRIXx Product Family
McDonnell Douglas	RAPIDS
NASA - JSC	Rapid Development Lab
Lockheed	LEAP
Honeywell	DSSA Toolset
Draper Labs	CSDL CASE System
Verilog	SAO + SAGA

United Technologies / Pictures-to-Code

United Technologies Corporation (UTC) 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, UTC has been able to decrease cycle time by approximately 40% while reducing errors by more than 60%. UTC 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, 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.

At each division, software development was driven by the standards of that division's 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 to satisfy the standards of their various customers, the 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 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

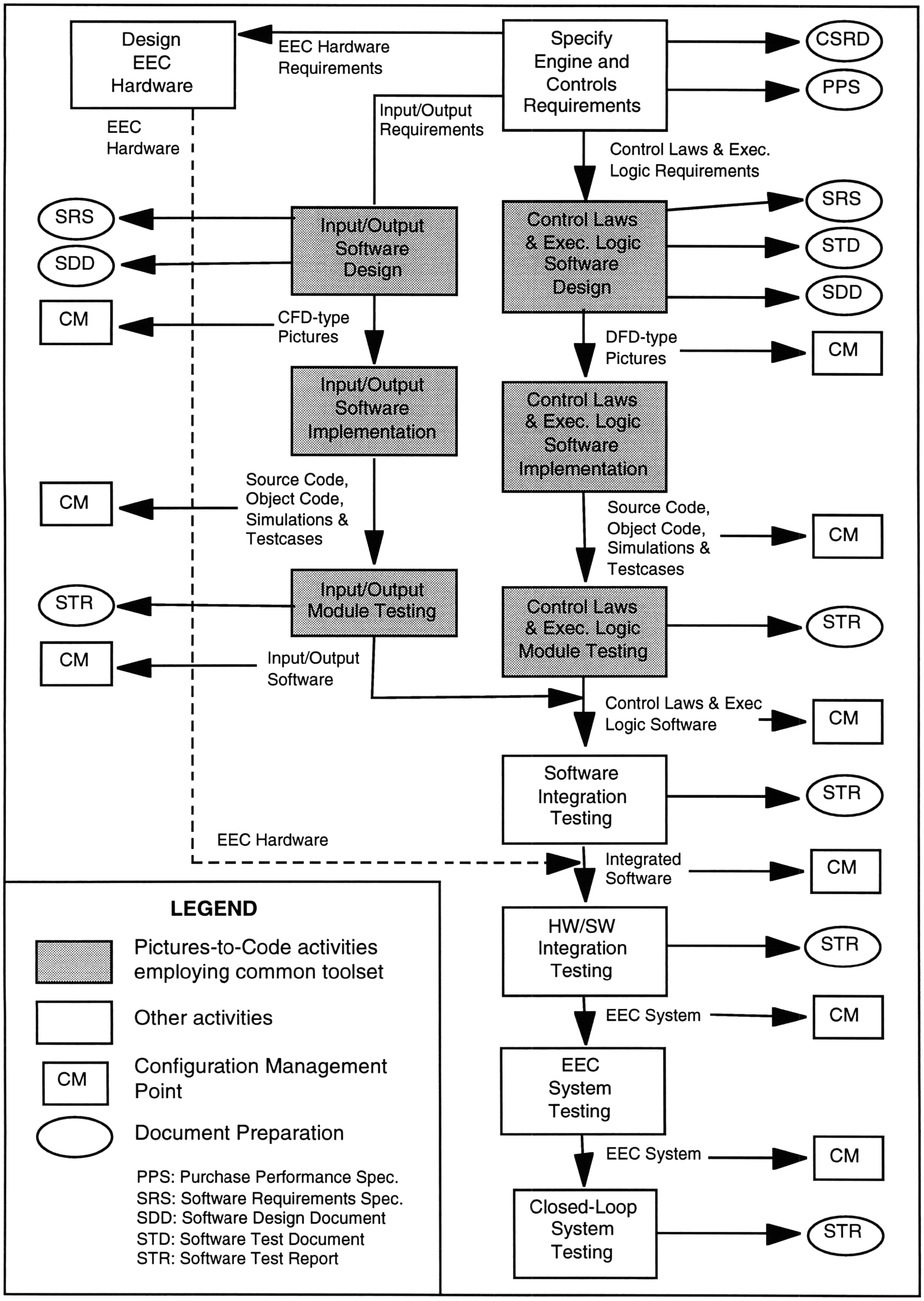

Design EEC Hardware
EEC Hardware Requirements
Specify Engine and Controls Requirements
CSRD
PPS
Input/Output Requirements
EEC Hardware
Control Laws & Exec. Logic Requirements
SRS
SDD
CM
Input/Output Software Design
Control Laws & Exec. Logic Software Design
SRS
STD
SDD
CFD-type Pictures
DFD-type Pictures
CM
Input/Output Software Implementation
Control Laws & Exec. Logic Software Implementation
CM
Source Code, Object Code, Simulations & Testcases
Source Code, Object Code, Simulations & Testcases
CM
STR
CM
Input/Output Module Testing
Control Laws & Exec. Logic Module Testing
STR
Input/Output Software
Control Laws & Exec Logic Software
CM
Software Integration Testing
STR
EEC Hardware
Integrated Software
CM
HW/SW Integration Testing
STR
EEC System
CM
EEC System Testing
EEC System
CM
Closed-Loop System Testing
STR
LEGEND
Pictures-to-Code activities employing common toolset
Other activities
CM
Configuration Management Point
Document Preparation
PPS: Purchase Performance Spec.
SRS: Software Requirements Spec.
SDD: Software Design Document
STD: Software Test Document
STR: Software Test Report

Figure 3.1 Overview of the Common Development Process for EEC Software

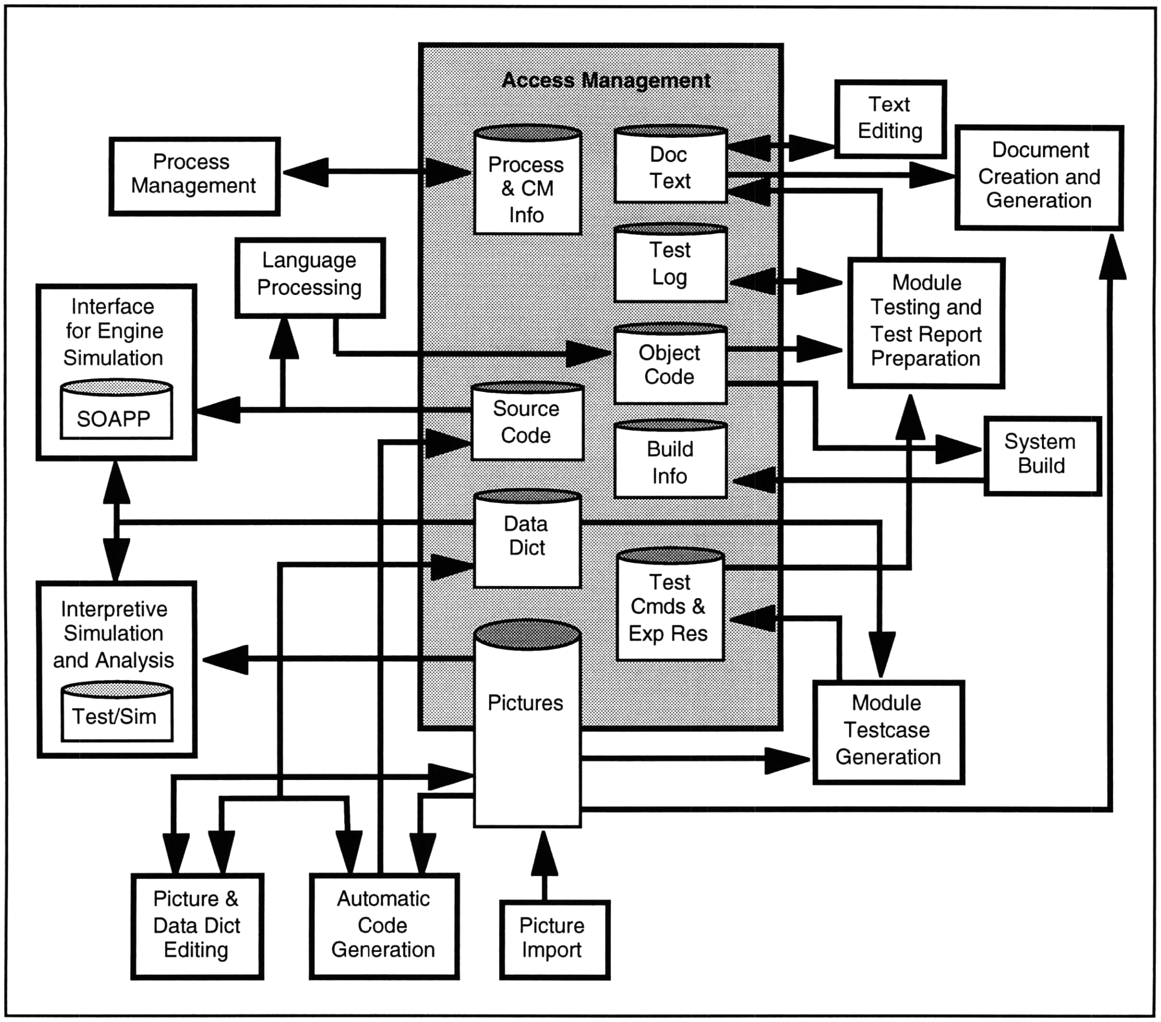

Access Management
Process Management
Process & CM Info
Doc Text
Text Editing
Document Creation and Generation
Language Processing
Test Log
Module Testing and Test Report Preparation
Interface for Engine Simulation
SOAPP
Object Code
Source Code
System Build
Build Info
Data Dict
Interpretive Simulation and Analysis
Test Cmds & Exp Res
Test/Sim
Pictures
Module Testcase Generation
Picture & Data Dict Editing
Automatic Code Generation
Picture Import

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

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

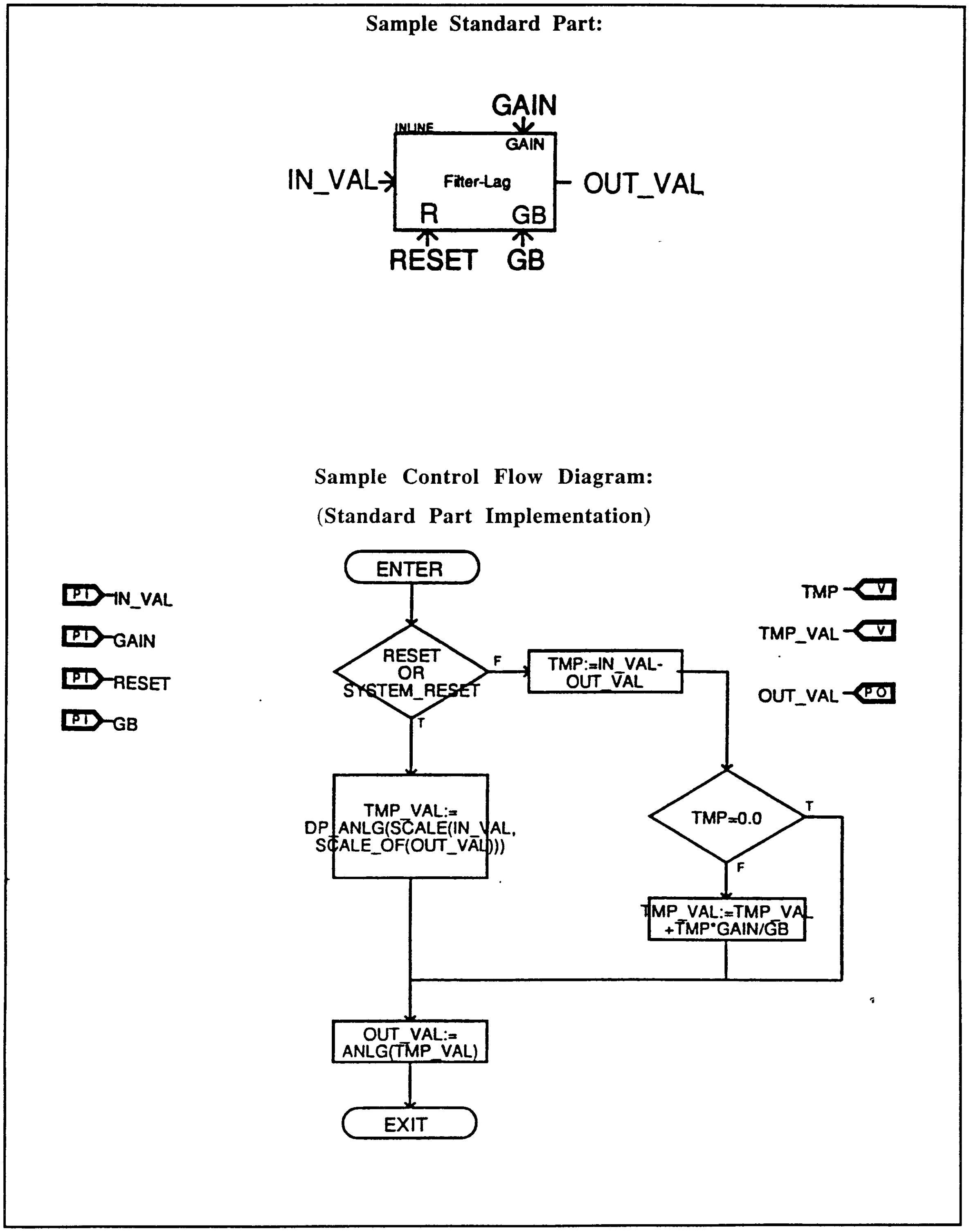
The core of the toolset is the automated code generator developed by Hamilton Standard. Hamilton Standard has been involved in controls systems since the 1960's and develops software for engine, flight and environmental controls. The automated code generator is based on a picture language. The picture 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).

An important element of the automated code generator is the Graphical Processing Utility (GPU). 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 comment headers and declarative statements and uses the design diagram to create the executable statements.

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.

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

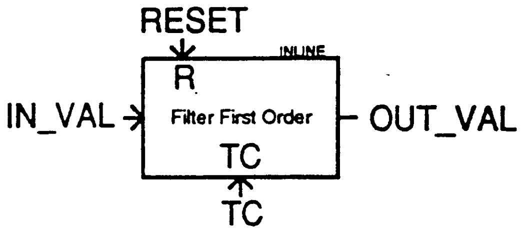

Sample Standard Part:

Sample Control Flow Diagram:

(Standard Part Implementation)

Figure 3.3 Control Flow Diagram and Standard Part

Sample Standard Part:



Sample Data Flow Diagram:

(Standard Part Implementation)

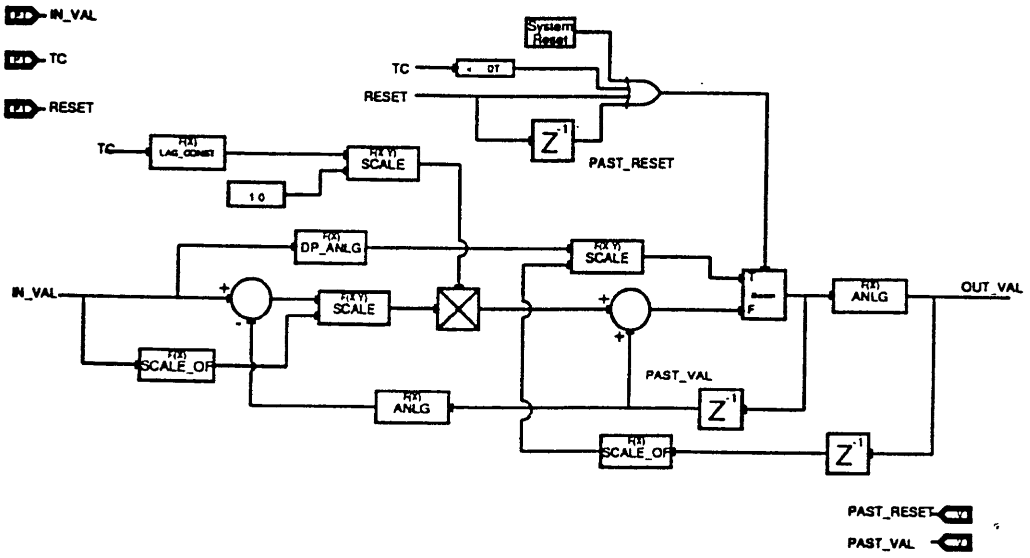


Figure 3.4 Data Flow Diagram and Standard Part

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 represent 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 (IRAD) 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 or 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.

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

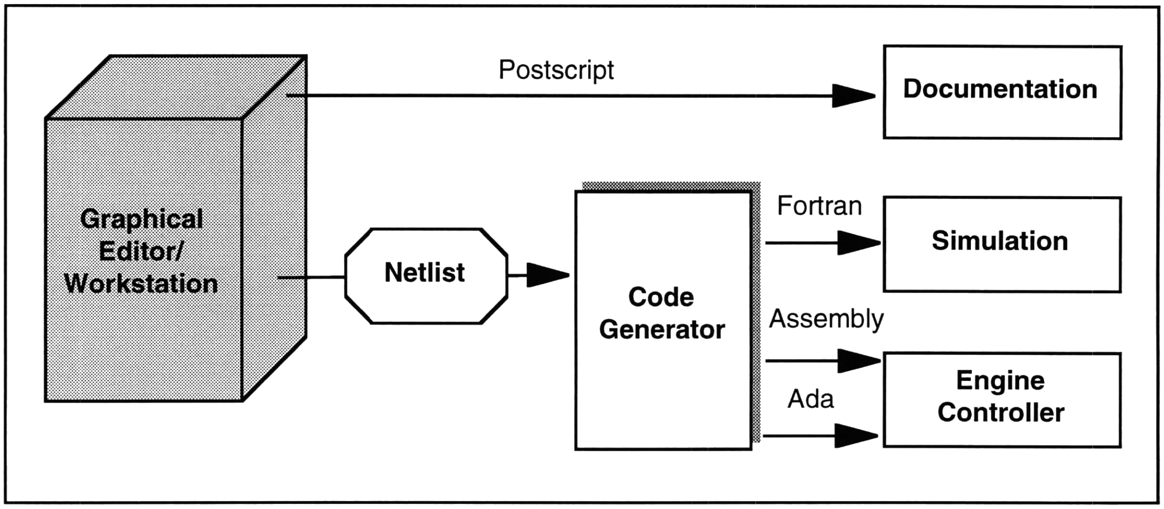


Figure 3.5 The BEACON Environment

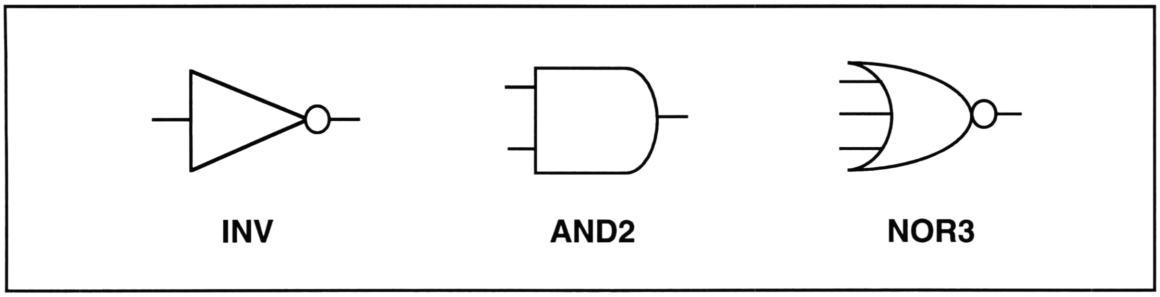


Figure 3.6 Sample Symbols from BEACON

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 multi-lingual, able to create FORTRAN, Ada and 68020 Assembly code. The same netlist can be used to generate code in any of these languages. The choice of target languages is being expanded to include C, ACSL, MATLAB and SIMULINK. The generator includes optimization capabilities to create efficient code. The choice of target languages allows the control system designer to either simulate the system or execute it real-time on the final processor.

Integrated Systems, Inc. / MATRIXx

Integrated Systems Incorporated (ISI), of Santa Clara, California, was founded in 1980 to develop design tool and operating system software for embedded systems. ISI focuses on six embedded systems markets: aerospace, automotive, communications, multimedia/entertainment, office/retail automation and industrial process control. ISI products include MATRIXx, which was first released in 1983. The MATRIXx product family offers a complete graphical solution for rapidly designing, simulating and testing microprocessor based control systems. MATRIXx was incorporated into the McDonnell Douglas RAPIDS environment and the NASA Rapid Develop Lab, both discussed later in this chapter.

The MATRIXx Product Family

The MATRIXx product family provides control system engineers with design, graphical simulation, automated code and document generation, and real-time hardware test capabilities. With the MATRIXx products, an entire control system can be graphically created and then modeled in actual operation. Control software can be automatically generated and downloaded to processing hardware for real-time execution. Complete documentation is also automatically generated to preserve the design and its data. The MATRIXx product family includes SystemBuild, Xmath, AutoCode, DocumentIt, and RealSim Series.

> 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.[9]

Xmath is an analysis tool used to evaluate and display the simulation results from SystemBuild.

> The AutoCode tool can automatically translate the block diagram representations into FORTRAN, C, or Ada source code. The source code can then be integrated with other interfacing software - software necessary for the code to run on the target processor such as a real-time operating system device driver, compiled and run on the AC-100 real-time computer to verify real-time and PIL performance.[10]

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) of Huntington Beach, California 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 (Riel, 1993). MDA-West

[9] 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.
[10] Ibid, p. 21.

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 often 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.

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

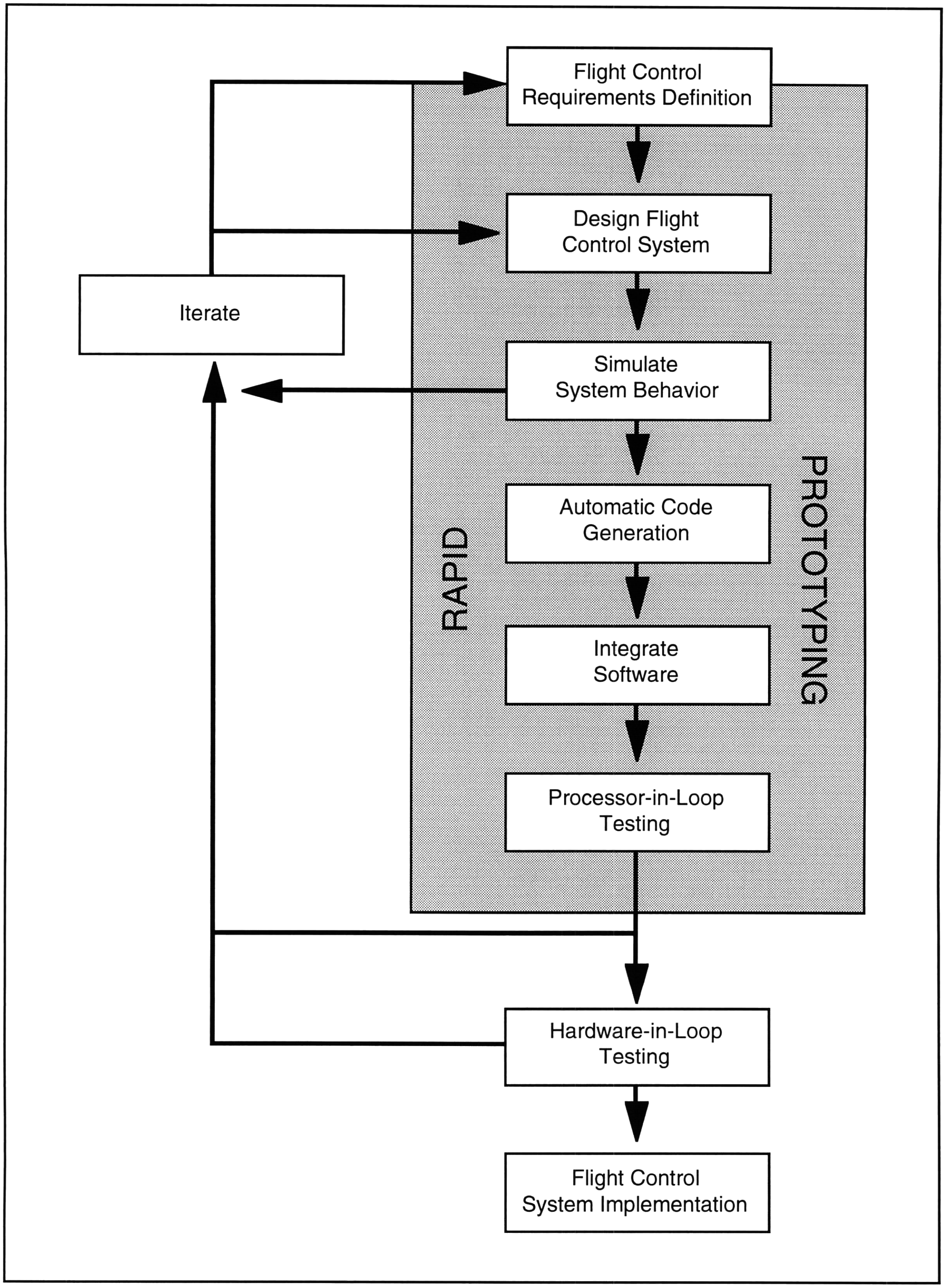


Figure 3.7 The RAPIDS Software Development Process

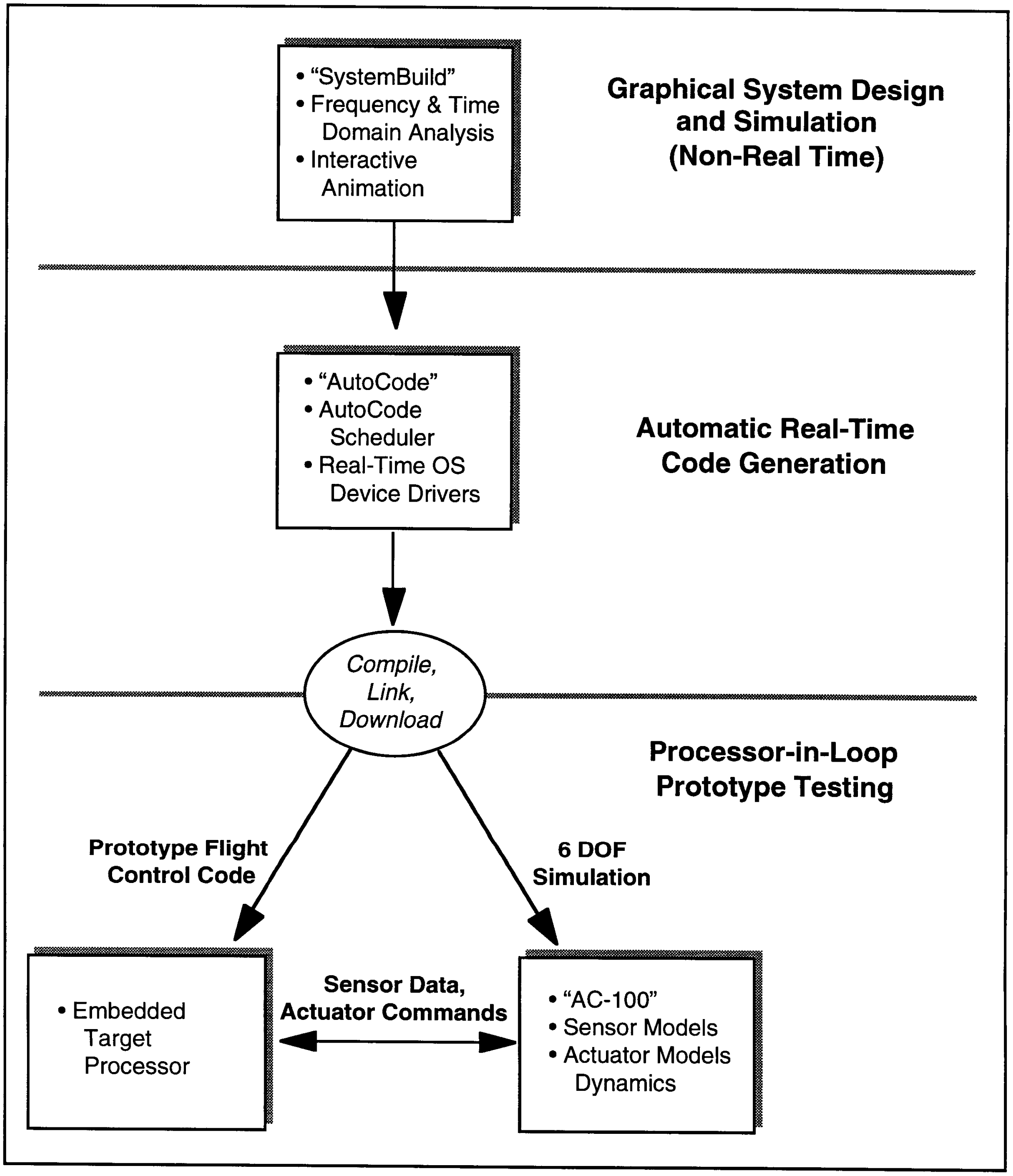

• "SystemBuild"
• Frequency & Time Domain Analysis
• Interactive Animation
Graphical System Design and Simulation (Non-Real Time)
• "AutoCode"
• AutoCode Scheduler
• Real-Time OS Device Drivers
Automatic Real-Time Code Generation
Compile, Link, Download
Processor-in-Loop Prototype Testing
Prototype Flight Control Code
6 DOF Simulation
• Embedded Target Processor
Sensor Data, Actuator Commands
• "AC-100"
• Sensor Models
• Actuator Models Dynamics

Figure 3.8 The RAPIDS Environment

NASA-JSC / Rapid Development Lab

The Rapid Development Lab (RDL) is a NASA Johnson Space Center laboratory created to explore and evaluate new technologies and processes for flight software and simulation development. Recognizing that the integration of CASE tools and rapid prototyping techniques can increase productivity and decrease both cost and schedule, 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 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 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 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

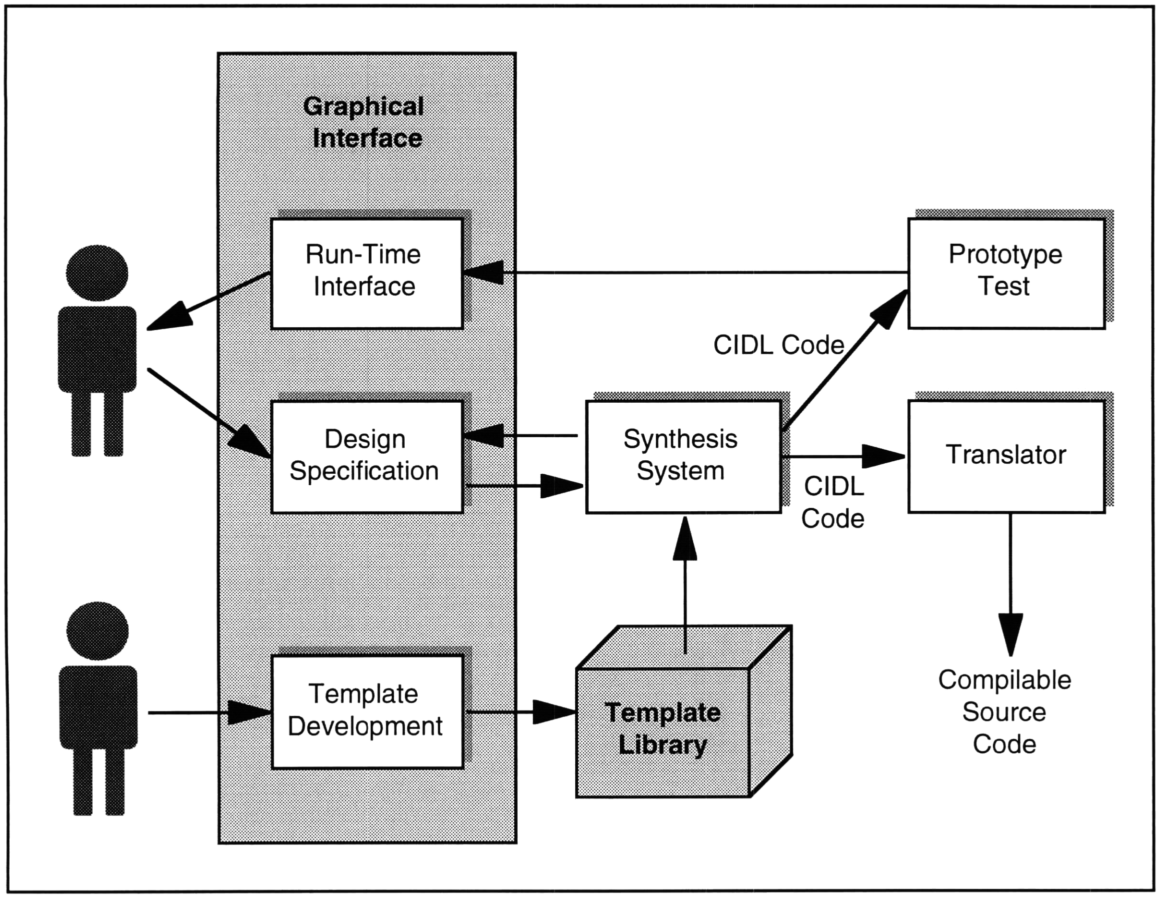


Figure 3.9 Lockheed Environment for Automatic Programming

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 translator is invoked. The CIDL code is translated 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 or 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.[11]

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

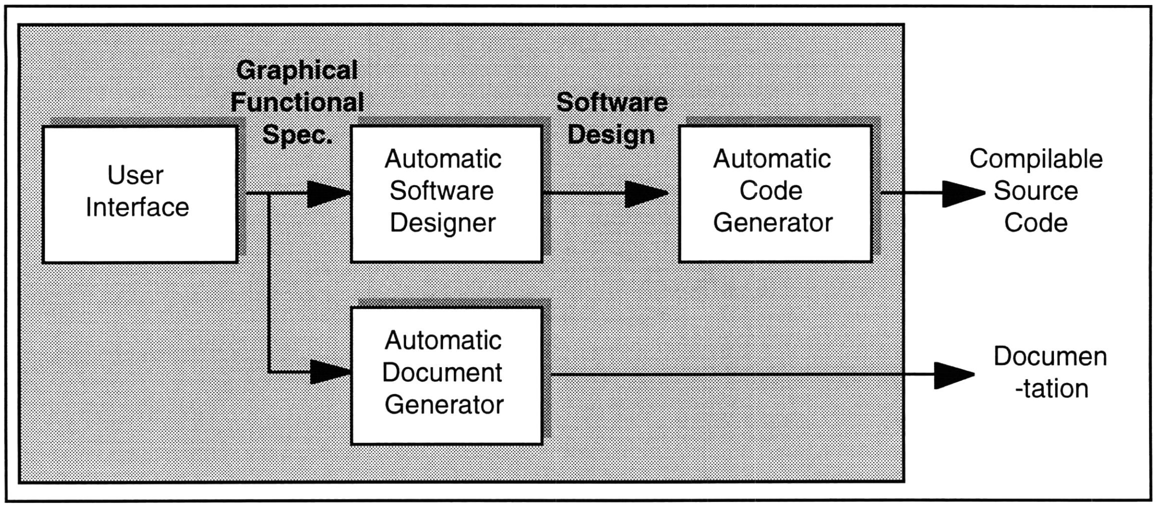


Figure 3.10 CSDL CASE System

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.

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 discusses industry experiences using software factory development and summarizes productivity improvements. 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 interviews.

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 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. This section 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 and during which activity it was found. These errors are process errors detected during verification of the software design and are corrected prior to formal release.

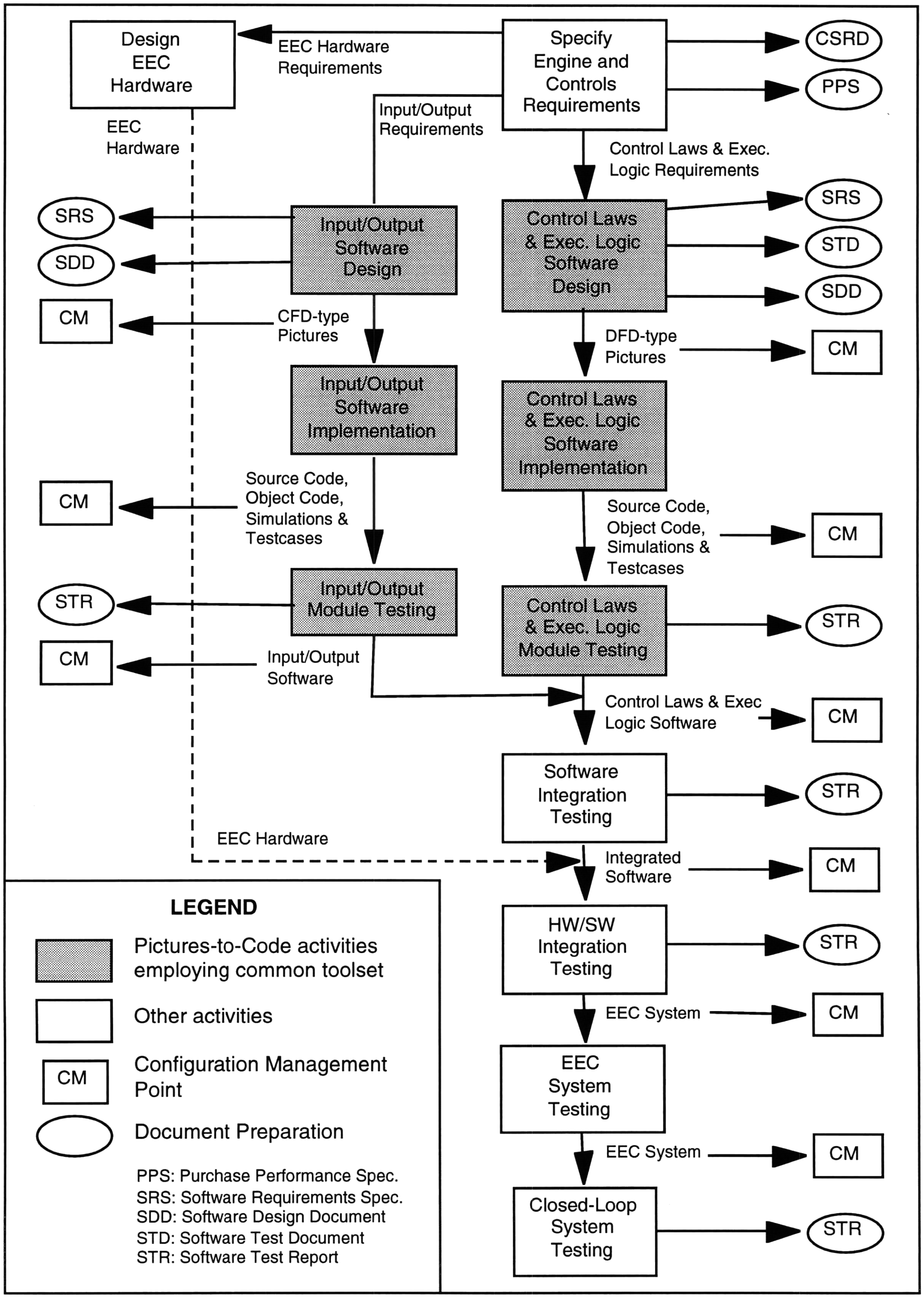

Design EEC Hardware
EEC Hardware Requirements
Specify Engine and Controls Requirements
CSRD
PPS
Input/Output Requirements
EEC Hardware
Control Laws & Exec. Logic Requirements
SRS
SDD
CM
Input/Output Software Design
Control Laws & Exec. Logic Software Design
SRS
STD
SDD
CFD-type Pictures
DFD-type Pictures
CM
Input/Output Software Implementation
Control Laws & Exec. Logic Software Implementation
CM
Source Code, Object Code, Simulations & Testcases
Source Code, Object Code, Simulations & Testcases
CM
STR
Input/Output Module Testing
Control Laws & Exec. Logic Module Testing
STR
CM
Input/Output Software
Control Laws & Exec Logic Software
CM
Software Integration Testing
STR
EEC Hardware
Integrated Software
CM
LEGEND
Pictures-to-Code activities employing common toolset
Other activities
CM
Configuration Management Point
Document Preparation
PPS: Purchase Performance Spec.
SRS: Software Requirements Spec.
SDD: Software Design Document
STD: Software Test Document
STR: Software Test Report
HW/SW Integration Testing
STR
EEC System
CM
EEC System Testing
EEC System
CM
Closed-Loop System Testing
STR

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. Both Pratt and Whitney and Hamilton Standard use the PtC toolset.

The PtC toolset has streamlined the software development 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 using the Graphical Processing Utility (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.

The PtC toolset has resulted in significant savings at the software design level. 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. Programmers typically attempt to avoid a large number of source lines per module in order to maintain intellectual control. With PtC, 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 using 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 than the engineers, 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. 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. With PtC, the standardized software design document 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 Figures 4.2 and 4.3 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 transitioned from 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.

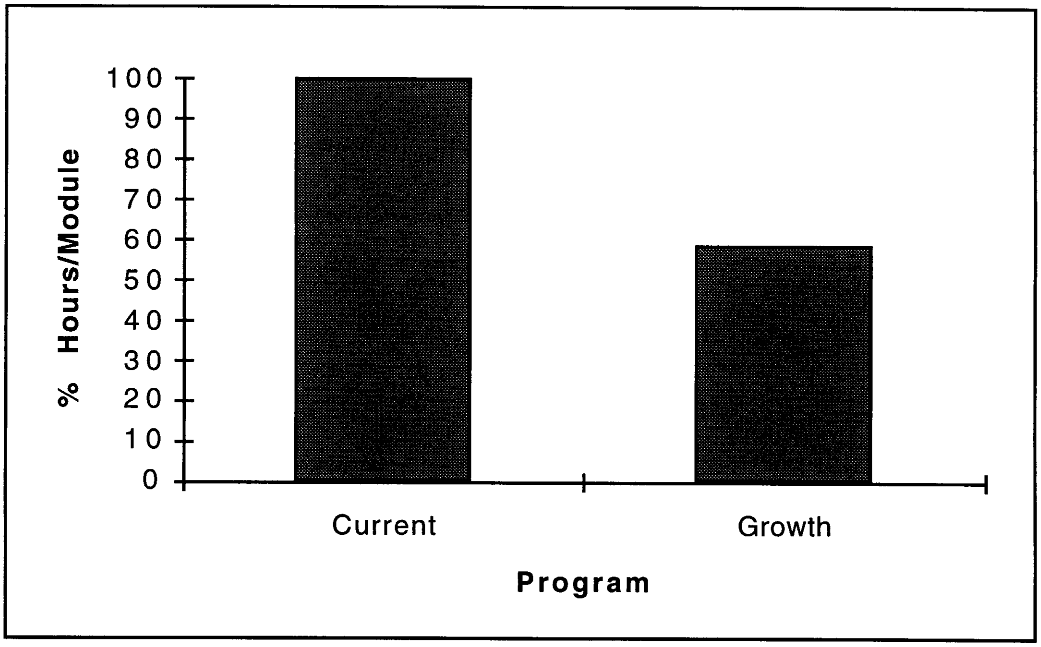


Figure 4.2 % Module Cycle Time, 4000 Current vs. 4000 Growth (Normalized to 4000 Current)

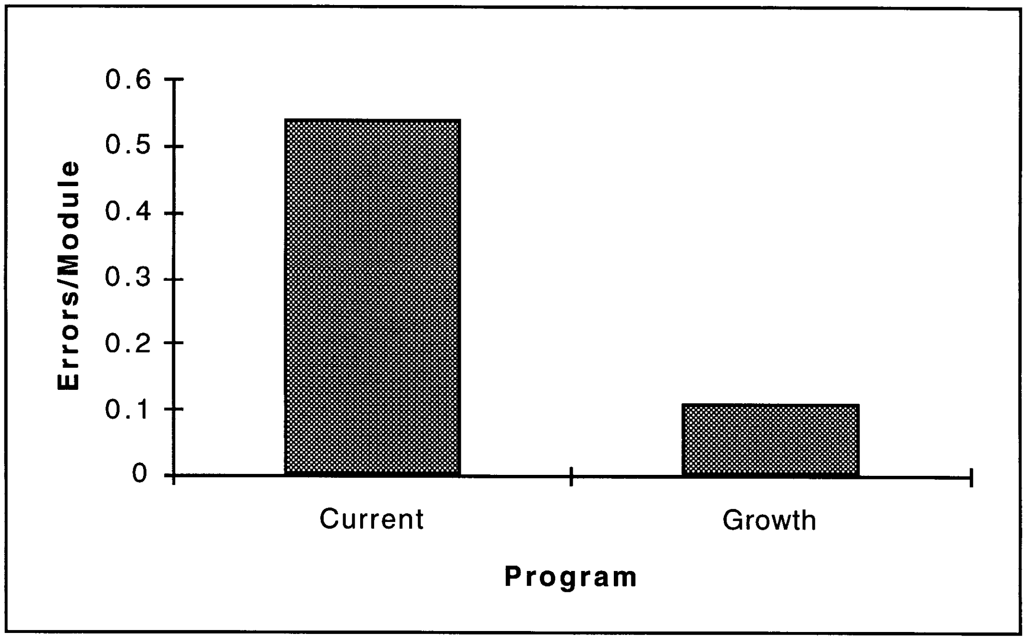


Figure 4.3 Detected Process Errors, 4000 Current vs. 4000 Growth

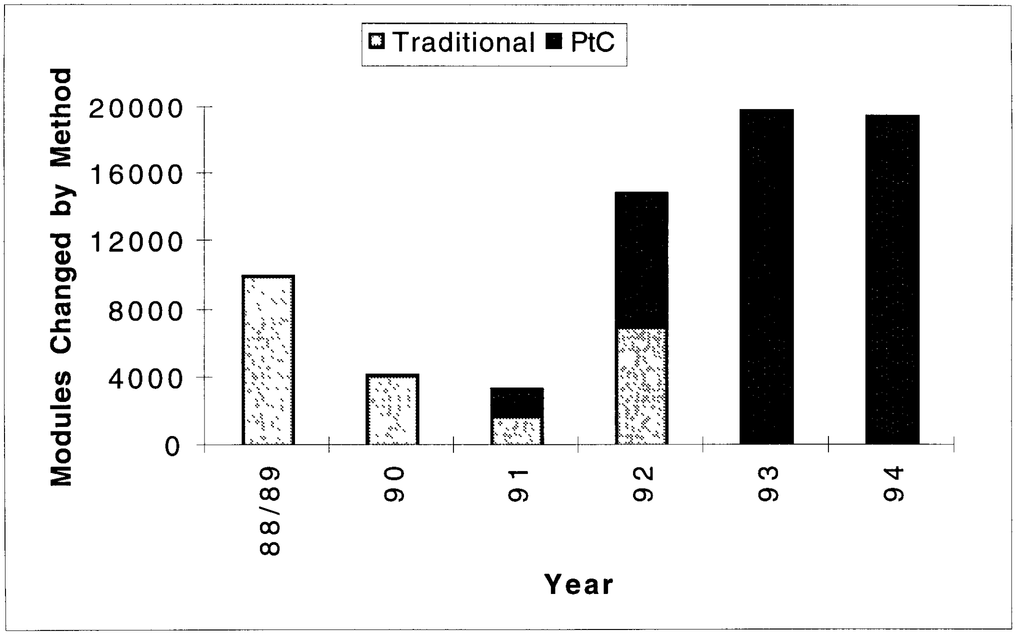


Figure 4.4 Modules Changed by Development Method vs. Year

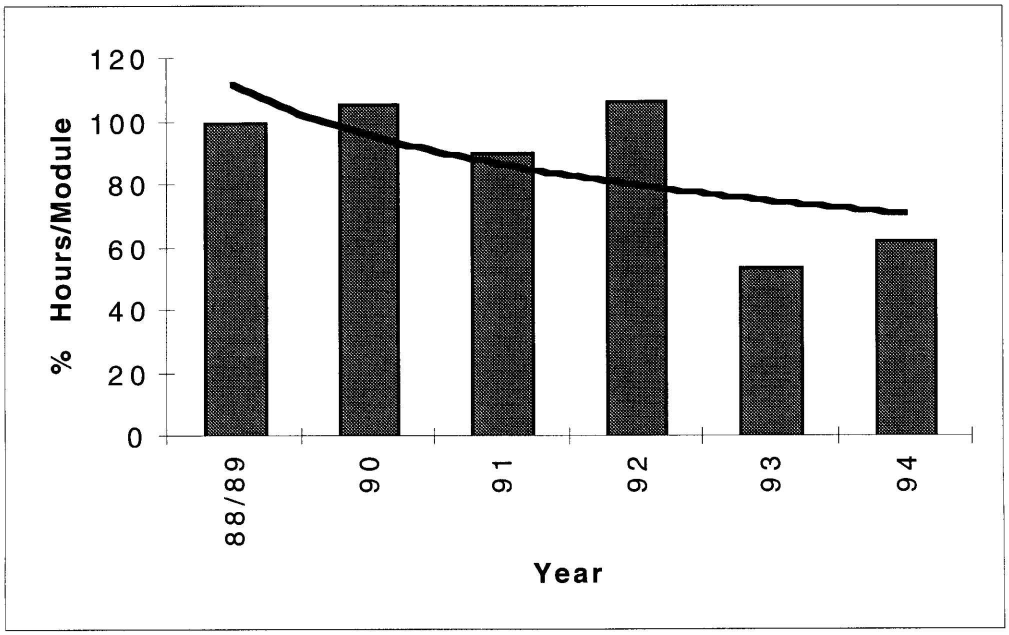


Figure 4.5 % Module Cycle Time vs. Year, All Large Commercial Programs (Normalized to 88/89)

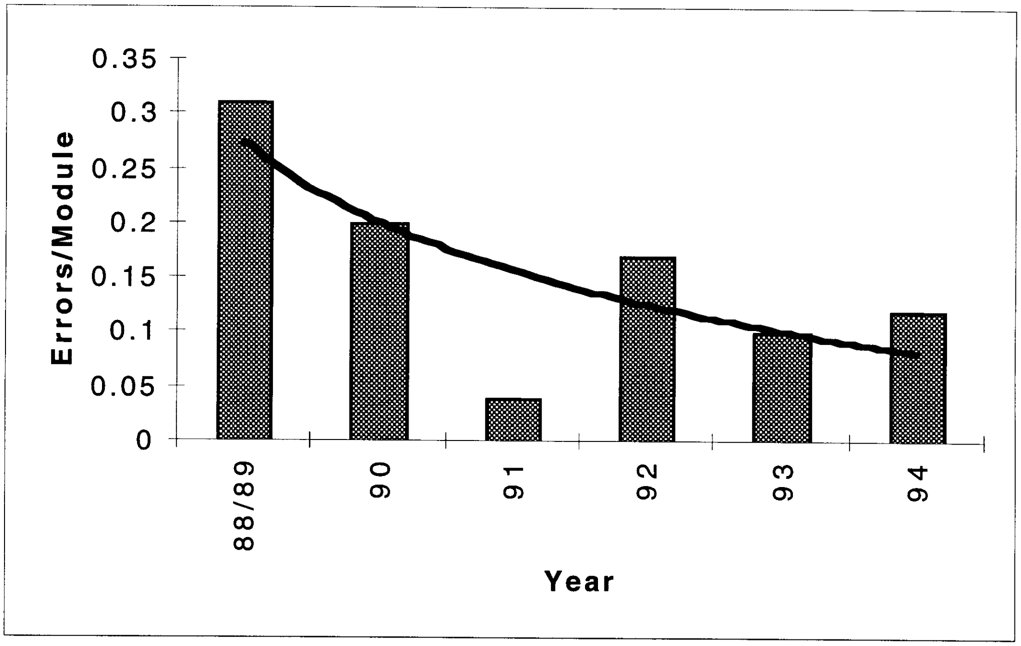


Figure 4.6 Detected Process Errors vs. Year, All Large Commercial Programs

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 with continuous process improvement initiatives updating both. 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 UTC the Software Engineering Process Group is responsible for process improvement and continually reviews both modification requests and 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 the percentages of errors detected at various stages in the process 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 that the majority of errors are detected during code read for the Traditional method and that the introduction of automated code generation virtually eliminates code errors, thus drastically reducing the errors found during code read and module test. This is expected since code reads and module tests verify the source code rather than 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. Hence with Pictures-to-Code, higher level functional testing is focusing attention on 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 early attention on requirements.

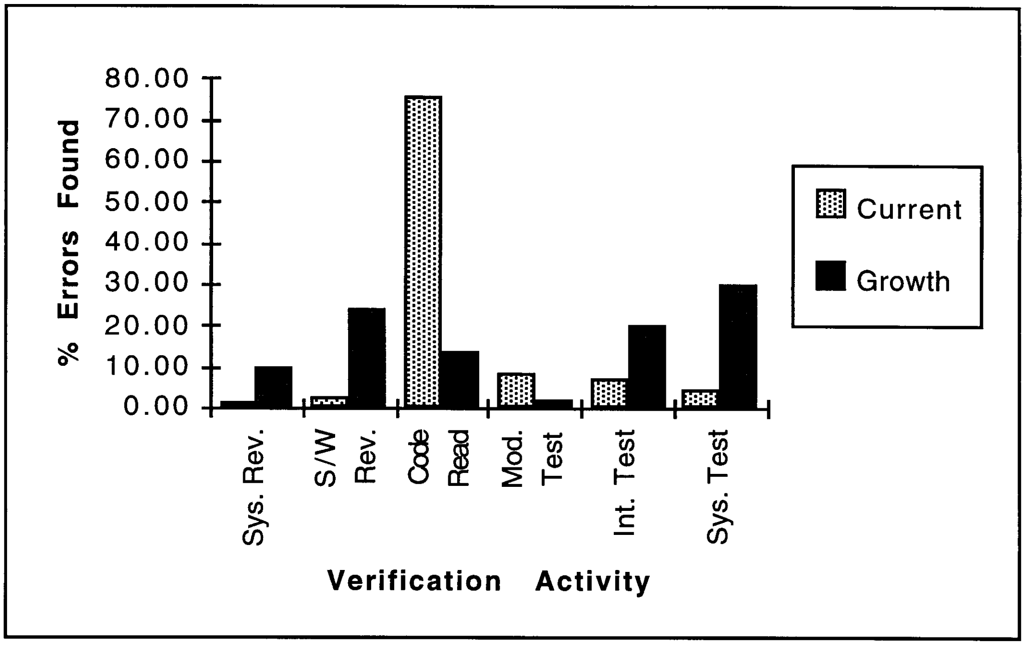


Figure 4.7 % Errors by Program vs. Verification Activity

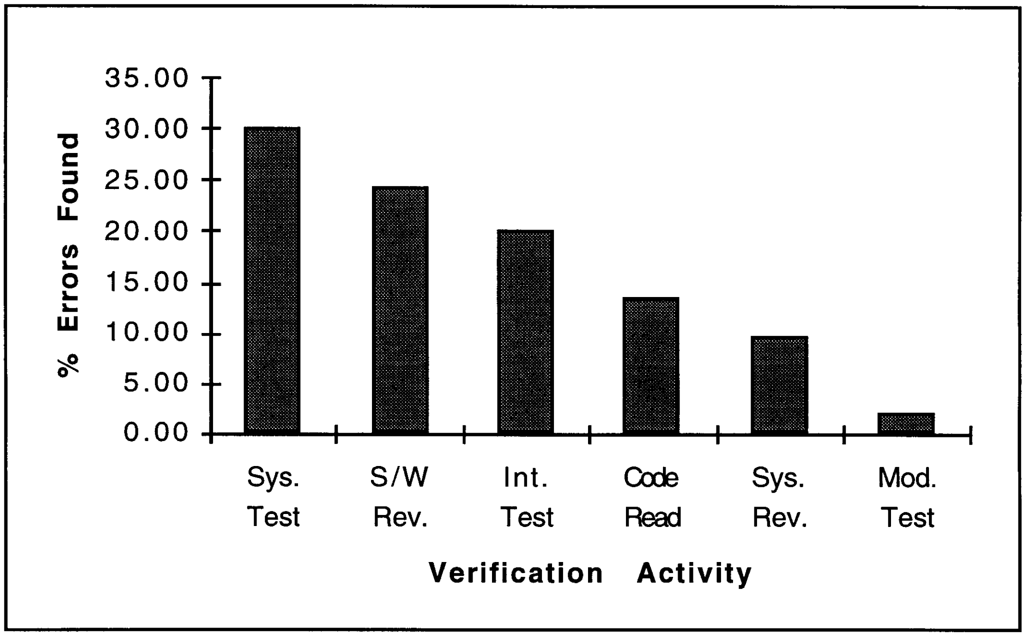


Figure 4.8 % Errors by Verification Activity for PtC Method

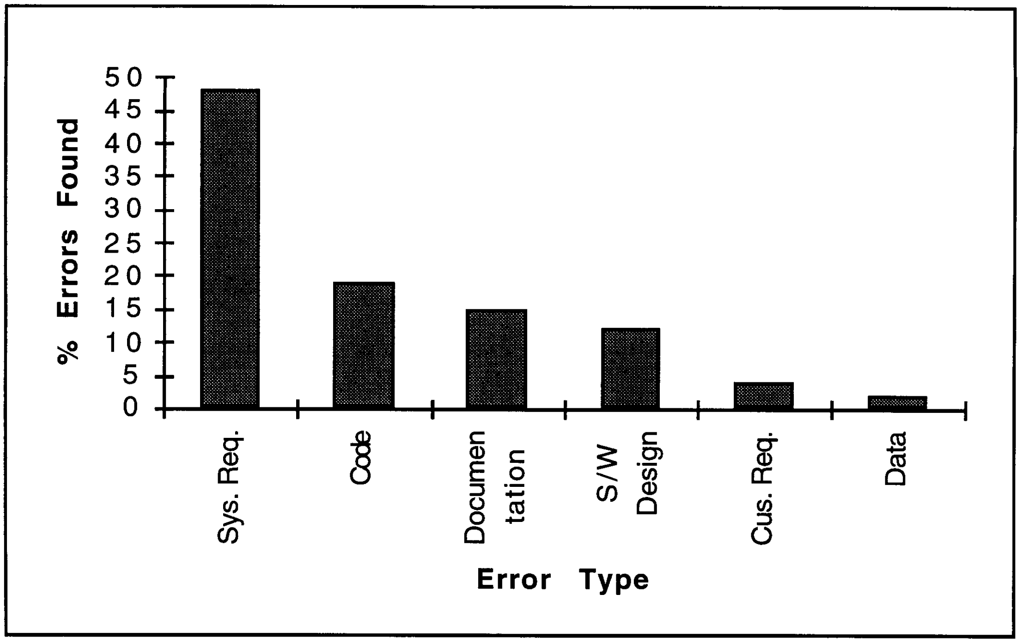


Figure 4.9 % Errors by Type for PtC Method

The Japanese Software Factories

The experience of United Technologies Corporation in establishing a common software development process and an integrated CASE toolset closely resembles the experiences of several Japanese software companies in establishing "software factories" during the 1980s (Cusumano, 1991). While each company had many reasons for starting their "software factories", they all shared the United Technologies' motivations to reduce the cost of software development and increase quality. These motivations are increasingly important as software becomes more complex and a larger number of system's functions are being implemented in software instead of hardware. Each company made a strategic decision to capture economies of scope by pursuing a standardized software development process. They invested heavily 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 automatic code generators. These management strategies allowed the companies to capture significant 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 companies to adapt to continuously changing customer needs.

The results for these Japanese companies and United Technologies are comparable in terms of 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 comparably small 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. 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		Activity:	
	Design	Implementation	Testing
Hamilton Standard	42%	32%	26%
Hitachi	38%	36%	26%
NEC	47%	37%	15%
Fujitsu	33%	30%	37%

NOTE: Data in Tables 4.1 through 4.3 compiled from Japan's Software Factories.[12]

[12] 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. General Electric, McDonnell Douglas, and Aerospatiale report productivity improvements of 35%, 25%, and 55% respectively:

> A productivity improvement of over 35% has been demonstrated.[13]
>
> Metrics tracked during the DC-X1 software development indicated productivity improvements greater than 25% can be achieved.[14]
>
> Aerospatiale has achieved a productivity gain of 55% in Airbus A320 design tasks.[15]

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 to which 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 185% increase to productivity was achieved.

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

[14] 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.

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

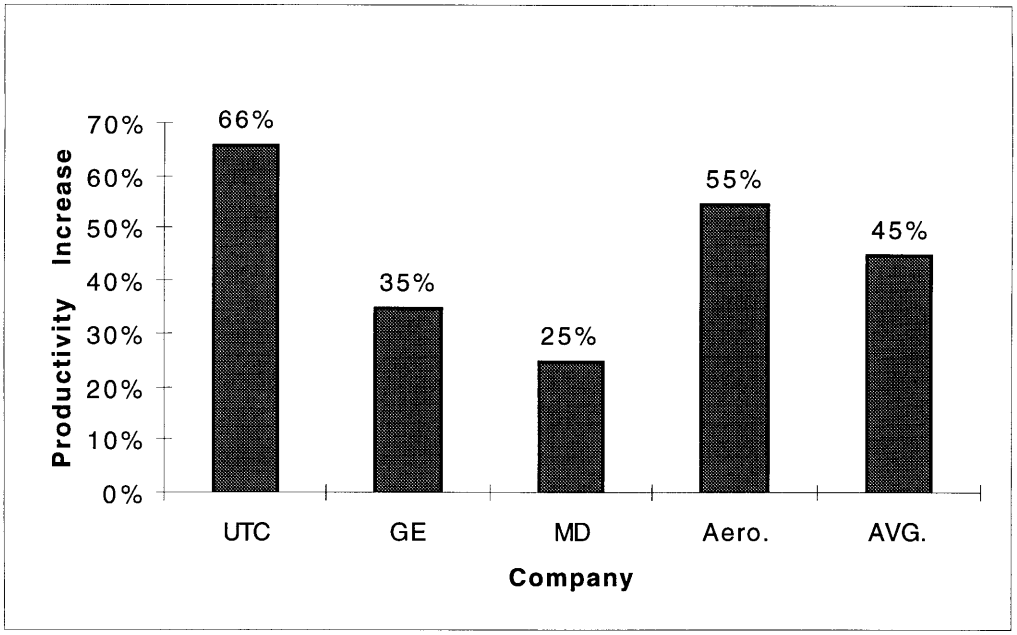


Figure 4.10 Comparison of Productivity Increases

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.[16]

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.

[16] 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. [17]

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[18]

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

[17] Baker, James; and Graves, Henson: "Simulation Based Software Development," Proceedings of the 8th Annual International Computer & Applications Conference, November, 1994, p. 123.

[18] Aichoun, Jean-Marc: op cit, p. 37.

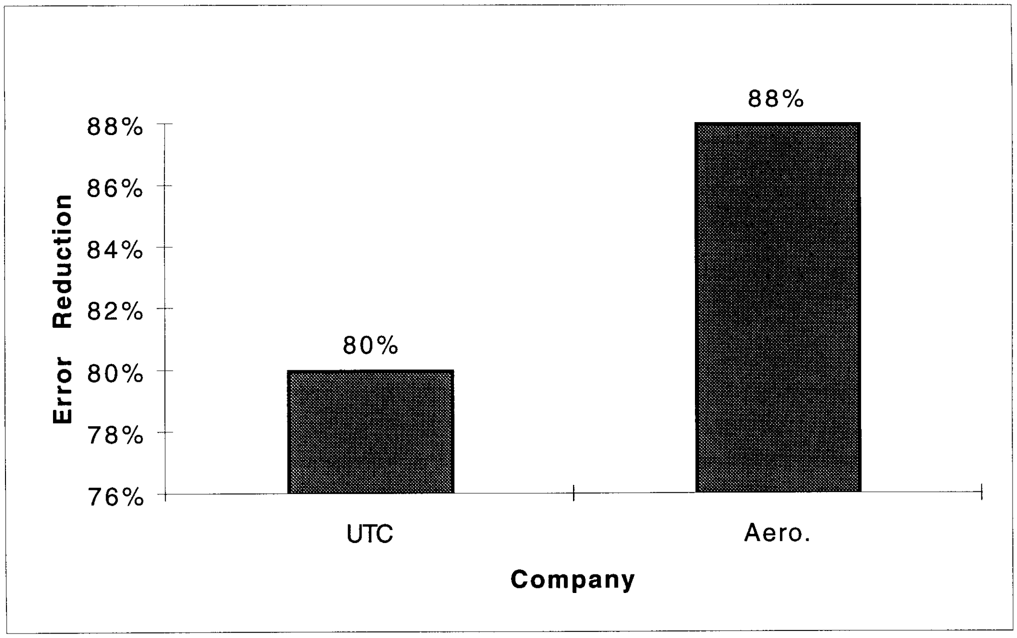


Figure 4.11 Comparison of Reductions to Encoding Errors

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 which can also be realized with software factory development are listed in 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
- Easier maintenance of software

The following excerpts from published papers and articles testify to these 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.[19]
>
> The automatic generation of code not only eliminates the time-consuming manual coding, but also avoids the manual introduction of bugs into the code.[20]

> The key to bringing down total development times is the ability to rapidly prototype the system. The engineers on the MSTI (Miniature Sensor Technology Integration) 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.[21]

[19] Dellen, Chester; and Liebner, Greg: "Automated Code Generation from Graphical, Reusable Templates," Proceedings of the 10th Digital Avionics Systems Conference, October, 1991, p. 299.

[20] 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.

[21] 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.[22]

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

[22] 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

THIS PAGE INTENTIONALLY BLANK

Chapter 5: Assessing the State of the Software Factory

Over the last six years there has been a growing base of experience in the aerospace industry that demonstrates the effectiveness of software factory development and software factory technologies. An approximate productivity increase of 45% can be expected and significantly greater gains are achievable in some circumstances, while at the same time increasing process and product quality. However, software factories have yet to see widespread adoption throughout industry, Table 5.1. For software factories to become institutionalized, industry leaders will have to keep abreast of trends in software development methods and technologies as well as business, policy, and cultural issues which influence the rate of change and direction of the trends.

Table 5.1 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
- Primarily used for demonstration projects, limited use on production programs
- Autocode technology is entering a commercialization phase
- Long term strategic management initiatives needed to create software factory

Diffusion of Software Factory Processes and Technologies

The 1990 National Defense Authorization Act included legislation mandating the creation of a critical technology report. The act, which defined critical technologies as "technologies essential for the United States to develop in order to further the long-term national security or economic prosperity of the U.S.", initiated the development of

Table 5.2 National Critical Technologies, Software and Toolkits Area

Sub-Area	Specific Technology	EP	NS
network and systems software	navigation and resource discovery tools	√	
	directories	√	
	registries	√	
	transparent embedding software	√	√
	operating systems	√	√
	run-time execution systems	√	√
	programming languages	√	√
	interpreters	√	√
	compilers	√	√
software engineering tools	computer-aided software engineering tools	√	√
	user-interface tools	√	√
	software testing tools	√	√
	IC design tools	√	√
software production	rapid prototyping	√	√
	modular/object-oriented programming	√	√

NOTE: EP = economic prosperity, NS = National Security

technology reports by various federal departments including the Department of Defense Key Technology Plan. This plan as well as the Office of Science and Technology Policy's National Critical Technology Report, which integrates all other reports, lists software and toolkits as a critical technology area. Each critical technology area is divided into sub-areas which are further divided into specific technologies. Several specific technologies within the software and toolkits area relate to software factories, Table 5.2 (OSTP, 1995).

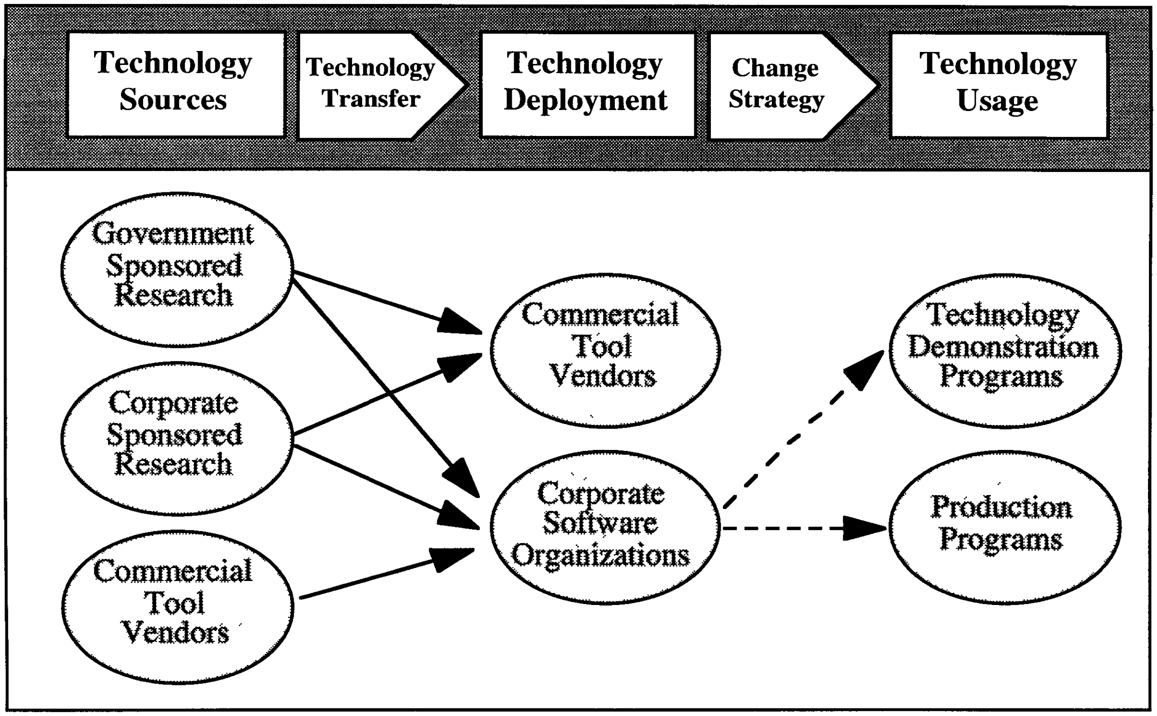


Figure 5.1 Software Factory Technology Diffusion Model

The critical technology reports are used to establish priorities for the Federal research and development (R&D) effort. However, R&D of new processes and technologies alone is insufficient to achieve the economic benefits of technology or lead to the widespread adoption of software factories by the aerospace industry. R&D must be accompanied by the transfer of technologies from R&D labs to industry software organizations and the implementation of change strategies by industry management. Figure 5.1 illustrates a technology diffusion model incorporating technology transfer and change strategy. The technology of automatic code generation makes an excellent example for illustrating the various paths of the model.

R&D programs sponsored by the government, the aerospace industry, and commercial CASE tool vendors form the many sources of software factory processes and technologies. The Domain Specific Software Architecture (DSSA) Program and the Software Technology for Adaptable Reliable Systems (STARS) Program are two of many

Table 5.3 DoD Programs Aiding Embedded Software Development

- Domain-Specific Software Architecture (DSSA)

  DARPA managed research program meant to transform the relationship between system and software engineers into one in which software engineers create building blocks and construction tools with which system engineers can create software subsystems.

- Software Technology for Adaptable, Reliable Systems (STARS)

  DARPA managed research program meant to increase software productivity, reliability, and quality by integrating support for modern software development processes and reuse concepts within software engineering environment technology.

- Comprehensive Approach for Reusable Defense Software (CARDS)

  Air Force sponsored program meant to develop an in-depth understanding of the paradigms, processes and methods needed to make reuse work in practice. CARDS provides training documents, consultation, and hands-on services including domain analysis, architecture definition, product line identification and scoping, repository development and acquisition support.

- Software Technology Support Center (STSC)

  Air Force Center meant to keep abreast of evaluated tools and tools under development which support the development of weapon systems software.

- Software Engineering Institute (SEI)

  Carnegie Mellon University operated Federal R&D center which develops models, frameworks, diagnostics, methods, standards, techniques, and architectures to help measure improvements in software engineering capability.

R&D Programs sponsored by the U.S. government through the Department of Defense, Table 5.3. Several aerospace companies have also demonstrated a strategic commitment to the software factory primarily through the investment in corporate R&D programs. For example United Technologies, General Electric, and Lockheed all developed their own automatic code generation technologies. The need for these corporate R&D programs was driven by the fact that commercial CASE tool vendors primarily support developers of business and information systems rather than developers of real-time engineering systems.

However, there are a few commercial tool vendors, such as Integrated Systems and Verilog, that have developed their own technologies in support of real-time systems.

These many sources of software factory processes and technologies result in multiple technology transfer paths. Technology transfer paths not only exist from the various technology sources to corporate software organizations but also to commercial tool vendors as the government and corporations look to third parties for maintenance of 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. These events indicate the beginning of a technology commercialization phase. As these various products compete with each other, capabilities will increase and costs will decrease.

Once a corporate software organization selects a new development process or CASE technology, a change strategy must be implemented. Change must be actively managed with a planned, predefined sequence of events. Currently the application of software factory processes and technologies has been mostly confined to technology demonstration programs. For example, automatic code generation has been used to develop flight software for one-of-a-kind aerospace vehicles like the Delta Clipper-Experimental (DC-X), the Miniature Sensor Technology Integration (MSTI) Satellite, and the Short Take-Off and Landing/Maneuver Technology Demonstrator (STOL/MTD, a specially modified F-15). Otherwise, use of automatic code generation on production programs has been limited. The product-line of aircraft engines is one exception. Software factories have been built by Pratt & Whitney and General Electric to develop electronic engine control system (EECS) software for both commercial and military aircraft. PW 4000 Growth, F-119, and GE-90 are all production engines with automatically generated EECS software.

Challenges to Software Factory Adoption

Implementing a software factory requires a change strategy that can successfully face several challenges. These challenges exist in the areas of technology, business management, and policy, Table 5.4. Although technology challenges are not confined to automatic code generation, the following uses the autocode example as a means to examine the general problem.

Table 5.4 Challenges to SW Factory Adoption

- Technology Maturity
- Technology Domain Coverage
- Technology Management
- Product-line Scope
- Software Ownership Rights
- Ada versus C

Technology Maturity

A computer program is a set of instructions for a computer. The most convenient means for a human to give instructions is through the human's natural language. Hence, a natural language, such as English, would be an ideal language for a computer program. But computers do not understand natural languages; instead they require binary inputs. However, machine language, assembly language, higher order languages (FORTRAN, Ada, C++, etc.), and very high-level languages (4th generation languages) illustrate that programming languages are evolving toward natural English. This evolution has been accompanied by the creation of language translation tools, such as an Ada compiler, which translates the higher order languages into machine language. Similarly, an automatic code generator is a language translation tool that converts a very high-level language, typically employing both diagrams and text, into a higher order language. The convenience of programming in very high-level languages comes at a price in terms of efficiency. When compared with code produced by hand for similar functions, code generated by language translation tools typically results in greater memory requirements and slower execution speeds. However, embedded software for aircraft and other real-time systems is often dependent on the efficient use of memory and, more critically, execution speed.

> Timeliness is the primary requirement for real-time systems. The activity of real-time control systems has to be synchronized with the dynamics of real world systems they control, which means that their activities must be performed within specified time intervals, and the data used for computations are valid only during a limited time after they are produced. Therefore, the dynamic behaviour of real-time systems must be predictable in a deterministic sense.[23]

The importance of execution speed and issues of code efficiency have affected the acceptance of automatic code generation because the efficiency of automatically generated code has not been acceptable high in many cases. However, automatic code generation technology continues to advance and optimization capabilities are improving. Also, issues with the maturity of autocode technologies are being offset by improvements in computer hardware. The speed of modern processors has been increasing at a dramatic rate with corresponding decreases in costs. Similarly, the last 15 years demonstrate that memory density doubles in approximately three years (Newport, 1994). As a result of these trends, a net reduction in product development costs can often be achieved with a systems approach which invests in greater memory and/or a faster processor in order to get the improved programming and testing productivity that comes with automatic code generation.

Further maturity in automatic code generation technology is also required to support team development and library reuse. Some of the commercially available autocoders require

[23] de la Puente, J. A.: “Real-time Software Development: A Perspective,” Proceedings of the 12th Triennial World Congress of the International Federation of Automatic Control, Vol. 2, July, 1993, p.693.

that all diagrams be resident in a single file prior to generating code, which can be a barrier to the decomposition of programs. The size and complexity of software programs for modern aerospace vehicles require the development of software by teams. Programs are decomposed into modules which are allocated to individual team members for development. The modules, in separate compiled files, are then linked to form the completed program. NASA-JSC's Rapid Development Lab overcame such a deficiency with MATRIXx by giving the role of "integrator" to one team member (Uhde, 1995). The "integrator" has the responsibility of copying the various diagrams of the team into a single file prior to code generation. This integrated diagram did not have a direct relationship to the individual diagrams of the team or reuse library, leading NASA to devise a special three tier configuration control process in order to use the MATRIXx tool with a development team and reuse library while maintaining proper configuration control.

Table 5.5 summarizes the desirable capabilities of a mature automatic code generator. As shown by the experiences of NASA's Rapid Development Lab, a successful strategy is to distribute the desired capabilities over several tools within the software factory support environment if the current autocode tool is lacking a desired capability.

Table 5.5 Desirable Capabilities for Autocoders

- Support of high level design

  Ability to create/link separate packages/modules

  Ability to create package/module headers

- Support low level design

  Ability to generate actual source code

- Support of module interface design
- Source code optimization
- Built-in simulation capability

Technology Domain Coverage

Coverage of software domains by automatic code generation technologies is a challenge that must also be addressed. The technology of automatic code generation has naturally developed within the guidance, navigation and control (GN&C) domain, where schematic block diagrams have historically been used to specify designs. 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 operational flight program. The current use of automatic code generation is confined primarily to the GN&C software domain.

GN&C is only one of numerous software domains comprising an operational flight program. Table 5.6 shows the distribution of the F-22 operational flight program by software domain (Aviation Week, 1995). The vast majority is general avionics software and GN&C comprises only a modest portion of the total. Either the current implementation of autocode must be shown to be applicable to these domains or new domain specific autocode technologies must be developed. Only then will autocode be applicable across many product-lines and productivity gains achieved across the aircraft industry.

Table 5.6 Distribution of F-22 Software Domains

OPERATIONAL FLIGHT PROGRAM	(100 %)		
Vehicle Management	4.1 %		
Utilities and Subsystems	7.1 %		
Avionics	88.8 %		
		Comm./Nav./Identification	26.6 %
		Electronic Warfare	14.4 %
		Mission Software	13.5 %
		Radar	12.4 %
		Controls and Displays	9.4 %
		Core	8.4 %
		Stores Management	2.1 %
		Inertial Reference System	1.8 %

However, much of the capabilities of existing code generators are likely to be applicable to other domains. For example, General Electric realizes over 80% coverage of a typical control system with their automatic code generation tool (Dellen, 1991). Of this coverage, Data-flow (or signal-flow) block diagrams comprises 35-30% and control-flow diagrams comprises 45-50%. Control-flow diagrams are traditional flow charts that capture a program's control structure and processing and should be applicable to all software domains.

Technology Management

The expansion of autocode technologies into new software domains will likely result in multiple domain-specific specification languages, which raises standardization issues. Standardization issues exist at both a micro and macro level. At the micro level, language standardization is the stringent definition of syntax (grammar) and semantics (what terms mean). Language standardizations are created by authorities such as the American National Standards Institute and the Institute for Electrical and Electronic Engineers for the purposes of eliminating multiple dialects and making the language portable. The graphical specification languages used by PtC, BEACON, and DoME/ControlH can be viewed as dialects of a single 4th generation language for GN&C systems and therefore could be standardized. This would allow a specification created in the PtC environment to be imported into the BEACON environment and translated into Ada without modification. But the fact that these languages are proprietary is a barrier to standardization.

Language standardization can also be implemented at a macro level by requiring the use of a single programming language across projects. In 1983, the Department of Defense pursued a macro level standardization by creating the Ada programming language and issuing DoD Instruction (DoDI) 5000.2, which included a directive to use Ada for all weapon systems software unless a waiver was granted. At one point some 400 higher order languages were being used in DoD systems (STSC, 1994) and the intent of DoDI 5000.2 was to reduce the cost of supporting these systems after deployment. This standardization may be undermined as new domain-specific 4th generation languages are developed. But since automatically generated code is easier to maintain, the ultimate goal of reducing support costs should still be attainable.

In addition to language standardization, there is also a debate going on in the aerospace industry about how much verification testing is required of automatically generated source code. Some take the position that the testing requirements for automatically generated code should not be relaxed. Others claim that unit level testing is not necessary, just as the output of a compiler is not tested as extensively as hand written assembly code. However compilers are validated. For example, the Ada Compiler Validation Criteria is a series of tests used to certify that compiler output meets the requirement of the Ada Language Standard (MIL-STD-1815). One possible solution to the debate is relating the required level of testing to the independent validation of code generators in a manner similar to what is done with compilers.

Product-line Scope

Issues affecting the adoption of a software factory are not just technical in nature. The business management aspects of the software factory include the definition of the product-line and investment in the supporting infrastructure and reusable assets. The costs of the supporting infrastructure (the development and maintenance of CASE tools, integrated environments, processes, libraries, domain models, etc.) are incurred at the business area level while the costs of developing reusable assets are incurred at the program or project level. It does take more effort to design reusable assets and time to populate a library. But an organization can realize a return on investment by maintaining a long-term view that cuts across multiple programs/projects to ensure that short-term expenses are offset by the larger cost reductions to be derived from automation, standardization, and systematic reuse.

Maintaining the long term view to realize a return on investment is easier if an organization can be sure of a market or demand for future development projects. Yet the market for future weapons systems is uncertain given the current environment of decreasing procurement budgets for the Department of Defense. However, existing weapon systems periodically undergo upgrades for insertion of new technologies. For example, the operation flight programs of aircraft are usually updated every two years. The increasing need to stretch the life of existing systems with retrofits should partially offset decreasing expenditures on new weapon systems development. To justify the investment in a software factory, an organization can also expand the scope of the product-line beyond weapon systems.

Expanding a product-line scope beyond weapon systems requires matching a military computer segment with its best fit in the commercial market. Obviously, product-lines encompassing military and commercial aircraft already exist. Software embedded in aircraft is real-time and often fault tolerant. Other commercial products requiring software with these characteristics include power plant control systems, medical monitoring and life support systems, and automobiles. Automobiles are quickly employing technologies first developed in the aerospace industry, including electronic engine control systems, GPS navigation systems, radar, head-up displays, and even drive-by-wire systems. However, an organization's ability to reuse software developed for a military weapon system on a commercial product requires that the organization retain software ownership rights.

Software Ownership Rights

Sections of the Defense Federal Acquisition Regulation Supplement (DFARS) document the criteria for determining the ownership rights to software. Ownership rights to noncommercial software and noncommercial software documentation is primarily determined according to who funded the development. Similarly, the Federal Courts grant copyright ownership of "work made for hire" to the commissioning party. The end result is that the Federal Government obtains unlimited rights to software developed exclusively with government funds and software documentation required to be delivered under contract. This includes the rights to embedded software for weapon systems, which can prevent contractors from reusing software components for other products. With unlimited rights and copyright, the Government is free to copy and distribute the software (even to other contractors) and a contractor would have to be granted permission from the Government before it could reuse software it developed. This is in contrast with the purchase of commercial software or a commercial product with embedded software (e.g. an automobile) where the developer retains the unlimited rights.

The Department of Defense Software Reuse Initiative, a focused activity aimed at establishing reuse as a common practice, has identified the issue of ownership rights as a barrier:

> Current acquisition policy indicates that the Government will balance its interests and the interests of industry and acquire only the minimum rights in computer software needed to accomplish the mission. For computer software developed under DoD contracts, the Government usually obtains unlimited rights. Exceptions include commercial software and software developed at private expense (which includes software developed with Independent Research & Development Funds). By acquiring unlimited rights to contractor-developed assets, the Government may be indirectly stifling the growth of a reusable software assets industry, discouraging widespread competition for DoD contracts (since many contractors are unwilling to sacrifice their rights to developed software products), and inhibiting the supply of trusted, reusable software products at a time when demand for such assets is rapidly escalating.[24]

The Software Reuse Initiative is currently studying the issue of ownership. Various levels of rights (i.e. unlimited, restricted, government purpose, etc.) may be allowed to be mutually agreed upon by the Government and contractor as "specifically negotiated license rights".

Ada versus C

Another challenge to the reuse of software components between military and commercial products is the choice of higher order language. Usually this challenge is reduced to one of Ada versus C.

> The most prevalent language for commercial software development is the C language, developed in the 1970s at Bell Laboratories to support operating system development. This language became very popular for

[24] Department of Defense: DoD Software Reuse Initiative: Vision and Strategy, 9th Edition, Document No. PD134.8, February 12, 1996, Section 3.3.

embedded commercial control applications because: C compilers produce compact, fast running code; C compilers are inexpensive (a few hundred dollars); the grammar of C is natural, which makes the language easy to learn; and nearly every software and electrical engineer fresh out of the university is proficient in C, which provides a large, stable work force.

These factors allow commercial software vendors using C to respond quickly and cost effectively to market changes, even on large projects with high turnover rates of engineers. The C language is rather clumsy for large projects, though, which prompted the development of the next generation language C++. C++ provides primitives for object oriented programming, a more abstract approach which allows further separation from detailed operation of the higher order language (HOL).

The problem for military software is maintenance. Several factors complicate maintenance, including criticality, complexity, and lifetime. ... In answer to these problems the Department of Defense developed the Ada Language standard, approved in 1983, to assure a standard HOL for operational code. Ada is structurally similar to Pascal, but more robust in the sense that compilers are required to check automatically for certain poor coding and design features which complicate maintenance. Although these features make Ada somewhat cryptic for most new programmers, the initial decrease in productivity is more than offset by cost savings during software support. Also, the increased complexity of Ada compilation systems forces much higher costs than C compilers ($30,000 versus $300). This cost difference, once again, is inconsequential to the operational savings for avionics programs. However, the high entry cost and shortage of skilled Ada code developers have slowed the acceptance of Ada in the commercial world.

... Ada will continue to emerge as the HOL of choice for both military and non-military large, complex developments. The basis of this choice is primarily supportability of such software designs. However, the policy implications of software reuse and commercial products will draw C and C++ software into this arena. This unification will accelerate as commercial C++ designers move to object oriented programming. For upgrades of existing platforms, there will continue to be a mix of HOL and assembly language since it is much too expensive to rewrite all the operational software for an aircraft.[25]

The challenge of Ada versus C is mitigated by the use of automatic code generation. Automatic code generators leverage the investment in compilers by translating 4th generation languages into higher order languages. By creating reusable components at the specification level and employing multi-lingual code generators, software developers can instantaneously implement an application in either or both programming languages.

The Role of Policy in Promoting the Software Factory

This research has shown that implementing software factories can reduce the cost and time required to develop software for embedded systems. However, current experience with software factories has been rather narrow and various challenges to software factory adoption exist. Policy can influence the rate and direction of technology development in order to overcome these challenges. The Department of Defense has implemented several policy initiatives addressing some of these challenges, including replacing DOD-STD-2167a with the more flexible MIL-STD-498 and creating the DoD Software Reuse Initiative. Support for these initiatives should be strengthened and their scope expanded.

Evidence indicates that successful implementations of software factories have involved a significant amount of in-house development to tailor the process and technologies to a specific domain of application. There is no commercial tool set currently available which encompass the entire range of needs for creating a software factory. Also, technologies developed to date have been confined to the GN&C domain. In order to expand the scope and use of software factories, the Department of Defense should actively support the research and development of technologies, such as new graphical specification languages and automatic code generators, and interface standards for tool integration. Since

[25] Newport, John R.: Avionic Systems Design, CRC Press, 1994, pp. 79, 80, 308.

software factories are focused around product-lines, and therefore do not encompass all software domains, care should be taken to avoid the development of a single all-encompassing language. Experience has shown that such an effort could prove to be a barrier by leading to unnecessarily complex and prohibitively expensive code generators. Also, tailoring specification languages to unique domains should facilitate the code optimization capabilities of code generators. The focus of DoD R&D support should be the development of non-proprietary languages and tools which will aid the technology diffusion process. And the R&D efforts should involve close cooperation with members of the aerospace industry and commercial tool vendors to ensure the necessary user buy-in and vendor support.

The objectives of a software factory are to achieve systematic reuse and capture economies of scope. These objectives are shared with the DoD Software Reuse Initiative and care must be taken to avoid potential conflicts between DoD software acquisition practices and corporate management practices. For example, the DoD should encourage contractors to maximize reuse but avoid requiring a contractor to meet a specific level of reuse. Rather, change strategies to incorporate reuse into management and engineering processes should be left to the contractor. Contractors can more accurately assess the availability of assets and their ability to tailor existing software components to new systems, which will drive the level of reuse. Time is needed for an organization to evolve a reuse capability and populate an asset library. The Department of Defense can promote and reward reuse by emphasizing incentives such as granting asset ownership rights.

The new MIL-STD-498 allows tailoring of the software development process from program to program. Although this will facilitate the adoption of software factory practices it will also increase the influence of program offices. It is important that DoD acquisition training does not reinforce mentalities, such as “not created for my program”, which can form barriers to software factory practices. Acquisition management needs to be more highly educated in the latest software development processes and technologies. Better teamwork between acquisition managers and contractors must be emphasized.

Areas for Further Research

Further research quantifying productivity gains needs to be conducted. Gains reported by GE, McDonnell Douglass, and Aerospatiale should be independently verified. Gains should also be mapped to the various phases of the software development life cycle. This thesis has shown that software factories exploit technologies such as automatic code generation tools to secure the near-term implementation cycle time reduction and quality improvement. Software factories can also generate longer-term savings by producing reusable assets suitable for future applications. Over time, an organization's asset library will grow and the level of reuse will increase. Since software factory development is an emerging practice, organizations utilizing software factory development need to be periodically revisited in order to monitor the growth of asset libraries and to develop an understanding of system design methods which best exploit reuse. Finally, data on the cost of implementing a software factory must be collected to support cost-benefit analyses and to calculate return-on-investment. Cost data should reflect capital investments in domain engineering, asset libraries and other supporting technologies. Benefits should account for reduced development costs due to improved productivity and decreased schedules and reduced maintenance costs due to lower levels of error creation and reuse of high quality components.

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.

de la Puente, J. A.: “Real-time Software Development: A Perspective,” Proceedings of the 12th Triennial World Congress of the International Federation of Automatic Control, Vol. 2, July, 1993.

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

Department of Defense: DoD Software Reuse Initiative: Strategic Plan, Document No. PD436.6, December 14, 1995.

Department of Defense: DoD Software Reuse Initiative: Vision and Strategy, 9th Edition, Document No. PD134.8, February 12, 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.

Henry, James E.; et al: "Using CASE Tools to Write Engineering Specifications," Proceedings of the International Society for Optical Engineering, Volume 1939, April, 1993.

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

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

Hopkins, Jay R.; and De Keyrel, Curtis R.: An Analysis of the Root Causes of Delays and Deficiences in the Development of Embedded Software for Air Force Weapon Systems, Thesis for the Degree of Masters of Science in Software Systems Management, U.S. Air Force Institute of Technology, December, 1993.

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^{2}E 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.

Office of Science, Technology and Policy (OSTP): National Critical Technologies Report, March, 1995.

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, 1993.

Riel, 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.

Software Technology Support Center (STSC): Guidlines for Successful Acquisition and Management of Software Intensive Systems, Version 1.1, U.S. Air Force, February, 1994.

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.

5778-86