



The Software Survivors

Software Engineering

by **Bill Cave & Henry Ledgard**

THE SOFTWARE SURVIVORS

by William C. Cave & Henry F. Ledgard

© Copyright 2005, 2006, and 2011 by William C. Cave & Henry F. Ledgard

Published by

Visual Software International
Spring Lake, NJ

December 2006

Revised Edition September 2011

TABLE OF CONTENTS

Acknowledgements	i
Preface	iii
Part 1 Software Needs, History & Objectives	
1 Surviving In A Competitive Market	1-1
2 Measuring Productivity	2-1
3 A History Of Programming	3-1
4 A More Recent History	4-1
5 Objectives of a Software Environment	5-1
Part 2 A New Technology for Software Development	
6 Evolution Of A New Technology	6-1
7 An Overview Of VisiSoft	7-1
8 Software Architecture	8-1
9 Architectural Design	9-1
10 Language Concepts	10-1
11 Data Structures	11-1
12 Rule Structures	12-1
13 Control Specifications	13-1
14 Simple Examples	14-1
15 Simulation	15-1
16 Very Large Scale Systems and Simulations	16-1
17 Managing Software Life Cycles	17-1
References	R-1
About the Authors	Back Cover

Acknowledgements

The authors would like to acknowledge the many contributors who provided ideas, assistance, and material that led to this book. This includes the development of key concepts, historical back up material, and the technology described here. The total number of people would be much too long to list, and likely would never be complete.

Many of the direct contributors are former as well as current employees of Prediction Systems, Inc. (PSI), the forerunner to Visual Software International (VSI). The twenty-four years of argumentation, experimentation, trial and change, using first hand experience on what works best - and what doesn't, the willingness to try new paradigms and approaches developed by others - to get it right or best from a productivity standpoint - were an overwhelming part of the culture at PSI and VSI. Without that very special culture, this technology would not exist.

We will mention a few of those who made significant contributions. Many of these people worked directly on the software, some as users shaping the approach. Much of the early software was developed by Dave Furtzaig, Pat Miller, and Marty Bopp. Early in-house users who provided design guidance were Hwa-Feng Chiou, Zong-An Chen, Don Lin, Dana Alexe, Rachel Marrotta, Luke Nitti, Jose Ucles, Kim Martis, Ken Saari, and Bill Wollman. Naeem Malik was instrumental in helping to install systems and train many users in Europe as well as the U.S., and ran the only real marketing organization PSI ever had. In more recent years, there were a number of students from Penn State, University of Toledo and Tongji University in China. Shunqian Wang, who managed the development team until recently, came to VSI from University of Toledo, and then brought students from Tongji University in Shanghai, e.g., Lin Yang and Yanna Jaing who still work with VSI, and Mingzhou Yang and Lei Shi. Shunqian's leadership was especially critical to the development of our graphical software, both in the CAD interface facilities, and with the run-time graphics. John Fikus provided major inputs as a user and developer of complex dynamic models, 3D graphics, and material for describing what VisiSoft does. Ed Slatt has provided contributions in the more esoteric area of parallel processing, as well as requirements, being a significant user building large scale planning tools. Dave Hendrickx and Manu Thaiwani have played key roles since the very early years, helping with analysis of design approaches as well as software development. As an outside consultant and user, Dave was the GSS User's Group Chairman for over a decade.

Special acknowledgement is due to Ken Irvine who has been involved in the concepts and basic design, both from a user standpoint as well as in developing the software itself. He has always questioned anything that appeared to be off-track, and has provided great help with our written material in general, as well as this book.

Development of many of the underlying concepts of VisiSoft would not have been possible without the first hand knowledge and analytical judgments from Bob Wassmer. He has been part of PSI, and particularly the VisiSoft development team, since the earliest beginnings, contributing as the key manager over all the years of this effort, as well as major designer and user.

Finally, contributions came from large numbers of clients of PSI and VSI, without whom none of this would exist.

Preface

This book is a combination of exact science and engineering observation. We treat two broad issues: the state of the US software industry, and a new technology to challenge this industry. We cover a number of different topics including competition, productivity, programming, object-oriented approaches, software environments, and software architecture.

Many recent reports show that software productivity in the U.S. has been in decline for at least the past 15 years. We hold that the root problem is the outdated technology foundation upon which current software is built. The software industry has followed a path of promises based upon assumptions that, when revealed, call into question the supposed scientific nature of the industry itself. It is our intent to show how the US software industry can quickly reverse this trend and achieve dramatic improvements in productivity.

This is also a book about a new technology that can make this happen. This technology is a clear departure from existing approaches. It does not require the steep learning curve of current approaches. It allows subject area experts to become good programmers. It places a premium on true software reuse. This means the resulting software must be highly understandable, to a level not seen in current programming environments. The differences in true reuse between this new technology and current approaches are dramatic. We provide cases that illustrate equally dramatic improvements in productivity.

Use of this new technology will require acceptance of innovative new paradigms, something historically shunned in the software industry. This predicament is described in one of the best books on new technology, “The Innovator’s Dilemma,” by Clayton Christensen. It describes the difficulty in understanding the cause and effect of driving forces in industries where inertia is very high and change is very slow, obvious characteristics of the software industry. In such an industry, innovation does not come about from within. This is because the major players have extremely high vested interests that may be toppled by a so-called *disruptive* technology.

Revolutions impose sweeping changes upon the frame of reference used to measure the values of a system under siege. They can impart significant improvements, wreak havoc on the system, or anything in between. Major revolutions are usually preceded by smaller waves, ones with much smaller force. When the big wave comes, everyone knows it has happened. For software, the major wave has not yet come - but it is on the horizon. This book presents a disruptive technology to the field of software. The technology itself has not come about by revolution, but by evolution - since 1982.

Our challenge is to convince the readers that this new technology is clearly worth any disruption that it will cause. In articles written by prominent people in the software field, e.g., [AN], [LAS], [LE2], [PO], there is a sense that everyone is getting ready to accept a major paradigm shift. Hardly anyone is disputing the need for it. Based upon observations, the disruption that a user encounters using the technology described here is small compared to the return on investment. The goal of this book is to help potential users understand why this technology works, and help make a long overdue change happen in the software industry.

Part 1

Software Needs, History & Objectives



Chapter 1 - Surviving In A Competitive Market

Freedom And Competition

U.S. politicians do a fine job expounding the virtues of free markets and the importance of world trade. That is, until it looks like another country is going to dominate a market for an important U.S. industry. Then the protection mechanisms start. The members of that industry start paying attention to what it will take to get the government to protect them instead of trying to become more competitive, or shifting their focus to other opportunities.

Very often, becoming more competitive requires unpleasant choices that certain groups perceive as not being in their best interests. Spending money to lobby, and fighting to garner votes for protection from other countries does not help to improve the basic infrastructure of the country to become a strong competitor. In fact, it teaches people the wrong lesson. Don't work hard to figure out better solutions than the other guy. Just get the government (read the taxpayers) to bail us out.

When people complain that other countries compete based upon cheap labor, they are really saying that the U.S. consumer has a better choice when buying the same, or even higher quality goods. This certainly makes life better for those who buy those goods. If the country producing those goods makes more money selling them to the U.S., it improves the lives of their own people. This happened in Japan. After enough years, lifestyles in Japan moved up the curve, to the point that their own costs of labor and everything else became unsupportable. This was due in part to government support for banks that in turn were supporting private industry beyond justified bounds. We now know the rest of the story. It all came crashing down.

Wal-Mart provides an appropriate example. Many are opposed to “outsourcing”. Yet, those purchasing items at Wal-Mart are asking “Is it better for me? Does it cost me less?” From an increasing number of purchasers the answer is “Yes!” Yet, the goods they buy were made in China, Japan, Mexico, or Brazil. Did that contribute to the loss of manufacturing jobs in the US? Of course! Is that a kind of “outsourcing”? Indeed! Yet, shoppers that are given total *freedom* to choose will keep purchasing at Wal-Mart.

So what’s the answer? We can stick our heads in the sand and hope the government will make it difficult for the importers. Or we can try to understand how to be a stronger competitor, and how we can take maximum advantage of our skills and the *freedom* to use them. Our best approach is to face the truth. If someone can provide goods or services of the same or better quality at a sufficiently lower cost for people to make a buy decision, so be it. To face the truth, we must ask ourselves if we can improve our productivity and do better in the foreseeable future. If the future is too far out, then we better look for another investment.

But we do not have to look outside the country to find those who would restrict freedom and competition. Many large organizations, including the U.S. Government, have tried to restrict the efforts of Microsoft. Microsoft lives without government help. That makes some politicians mad, but historically it is the best way to serve the country.

Microsoft is where it is because it produces what people want. Many people dislike Windows and the Office applications that go with it. They use Microsoft’s products because of their own need to be productive - to be competitive. Many Microsoft products have become a defacto (real) standard - not licensed or imposed by government decree. This helps everyone become more productive.

WHAT FOSTERS PRODUCTIVITY?

Productivity is directly affected by the way people view their job security and the protective or competitive nature of the work environment. The differences are considered below.

Protective Environments

A protective environment is typically characterized by attempts to reduce real measures of output or productivity. For example, piece work (getting paid by the number of pieces you produce) is not allowed in a union environment. Such measures of value are now taboo, being effectively banned by the government. Another characteristic is being protected by who you know. Protective environments are typically political in nature - they depend on “who you know”. In such environments, survival does not depend upon the number of points you can score for the team (that would imply a competitive environment). The only score kept is the number of favors one owes.

This leads to another characteristic - irreplaceability. This is imposed by hidden or secret processes that only one or two “inside” people know. Sometimes it is born out by exotic rituals that outsiders have difficulty relating to. In the end, person A cannot understand what person B has done to get ahead, rendering person B irreplaceable.

Competitive Environments

In competitive environments, everyone must be a team player. There is no time for politics or rituals that waste time and energy, and produce no direct results. To ensure that production is strong and stable, parts and people must be quickly replaceable so they are independent of the process. Standardization is set to help - not stymie - people's understanding of what each other has done, so one can easily take over from another. If one person discovers a way to improve productivity, it is quickly shared among those on the team so everyone can score higher. Being able to communicate ideas quickly is critical to survival in a competitive environment.

The Business Environment Versus The Work Environment

We must also distinguish between the business environment (sales and marketing), which may be very competitive, and the work environment (production), which may be protective. If company A views the business environment as very competitive, and creates a work environment to match, and B harbors a protective work environment, then A will move ahead of B. There are many examples of has-been companies that found themselves unable to compete in their line of business as competitors moved into their market. Such companies typically had it easy before the competition came along, so the work environment was protective. Unless the business environment is constrained, typically by politics, the most productive company will eventually prevail - just as in sports.

THE SOFTWARE INDUSTRY

So what does this have to do with software? Everything. The number of hardware devices that do not depend upon software is shrinking everyday. The number of devices using computer chips is growing everyday. As we build more computational power into these devices, they do more for us. They help us to become more productive, and therefore more competitive.

Software is likely to be a critical industry for any country. It is already a major factor in determining the productivity - and thus the competitiveness - of nations as well as industries. We encourage those within the software industry to seek the truth on this important topic, for all is not well.

Productivity In The Software Industry

Let us look at the U.S. software industry. In a September 1991 Business week article titled "Software Made Simple," [19], industry experts offered great hope for productivity gains in the future with Object Oriented Programming (OOP) technology. The article admitted that there were naysayers who compared OOP to the promises of Artificial Intelligence (AI). But it was stated that, unlike AI, object technology would have an immediate and practical payoff.

In 1995, leading technologists John Warnock (CEO of Adobe) and Gordon Bell (architect of the DEC VAX) were quoted in Upside [97] as saying that OOP is a disappointing technology, that it does not deliver real improvements in software productivity.

In the July 1996 issue of Software Developer & Publisher magazine, an article by Cave [25], "Software Survivors," analyzed the reasons for declining software productivity. It quoted two other articles, [18] and [86], showing that, while productivity in the computer hardware industry was *increasing* faster than in any other industry, software industry productivity was *declining* faster than all others for the period 1990 - 1995.

One of the quoted articles was a 1995 Business Week issue, [18], that surveyed productivity in 25 industries. From this article, Cave derived percent productivity change over the previous five year period and came to some astounding conclusions. These are shown in Chart 1-1. Productivity changes in chips were at the top of the productivity list (+153%), and Software was dead last (-11%).

Independently, in February 1995, the Standish Group published a report, [SG], on the software industry supporting the negative productivity findings and describing software failure statistics. When discussing these negative results with higher-level managers responsible for funding software projects, the managers agreed with the data, saying it matched their experience.

In a March 23, 2003 press release [87] on software, the Standish Group noted that "Project success rates have increased to just over a third or 34% of all projects." Can you imagine such poor statistics in any other industry but software? But even that success came at a price. They also said that "Time overruns have increased to 82% from a low of 63% in the year 2000. In addition, this year's research shows only 52% of required features and functions make it to the released product."

A December, 2004 article by Robert Groth, published in IEEE Software, [44], showed the percent productivity gain *per year* of various major industries over the 1998-2003 period, see Chart 1-2. Again, over this period, computer chips (up 95%) had the most gain in productivity. Software was again last on the chart with a decline (down 5%). The general productivity issue is discussed in Groth's article, where different views of this dilemma are offered.

Some would argue that software applications are becoming increasingly complex, and when taking complexity into account, simple statistics such as those in Business Week do not convey a true picture. Thus we have heard arguments that denounce these numbers, e.g., arguments noting that we are attempting more sophisticated applications. This factor alone might account for negating a portion of the gross measure of productivity.

However, a January 15, 2004 article on the 2004 CHAOS report by the Standish group [88], indicated that software project success rates improved over 10 years, stating that "The primary reason is that projects have gotten a lot smaller." Another reason given was that "People have become much more savvy in project management."

CHART 1-1. Data From Business Week - January 9, 1995

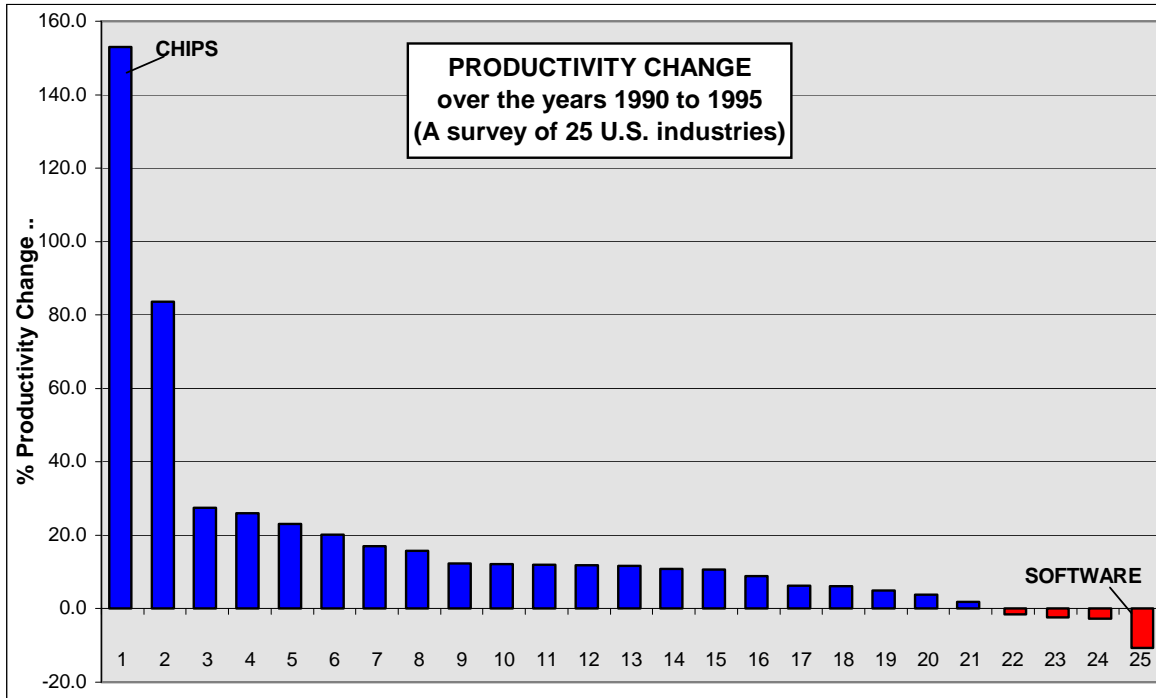
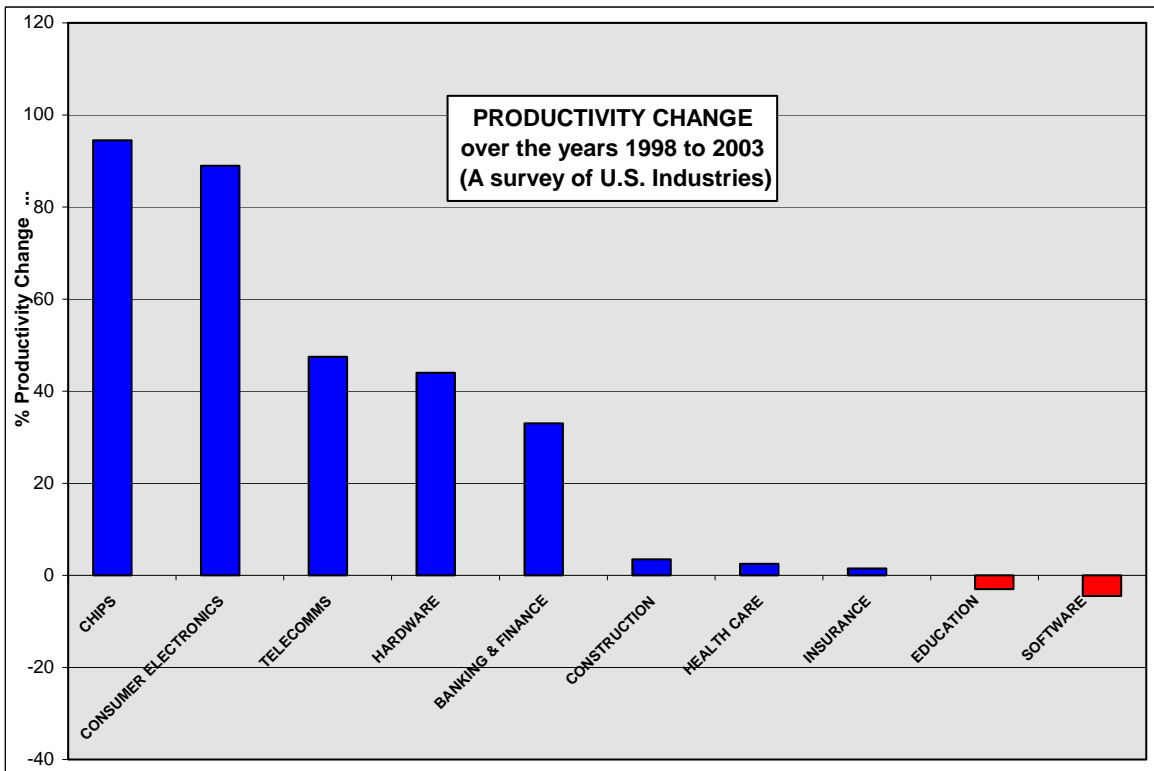


CHART 1-2. Data From Groth [44], IEEE Software, November/December, 2004



Additionally, one can make arguments that the amount of memory available, the speed of hardware, the documentation facilities, the editors, and automatic control and test facilities compensate for any increase in complexity. However, none of these arguments have a scientific basis. Our experiences, as well as those whom we have queried in the business of producing software, support the observations about the low productivity of current software technology. But none of these observations seem to stop the software industry from following a path that clearly puts it further behind in the productivity race each year.

An article in the November 2003 issue of the Communications of the ACM, "Measuring Productivity in the Software Industry", [1], discussed the productivity problem in software. But it did more than just report on the problem. It developed a rationale for measuring the productivity of development and support environments as a first step to solving this problem.

As we approach what *appear* to be physical limits in the semiconductor industry, engineers have continually removed the barriers. Through the use of Computer-Aided Design (CAD) tools and graphical visualizations, the promise of Moore's law continued to be fulfilled for many years. Microprocessor speed, memory size, and affordability skyrocketed. As stated by Larry Constantine, [30], software engineering has capitalized on these advances to offset its own poor productivity. But in recent years, Moore's law has not held. Improvements in hardware speed have been slowing dramatically. And this is putting more pressure on software productivity.

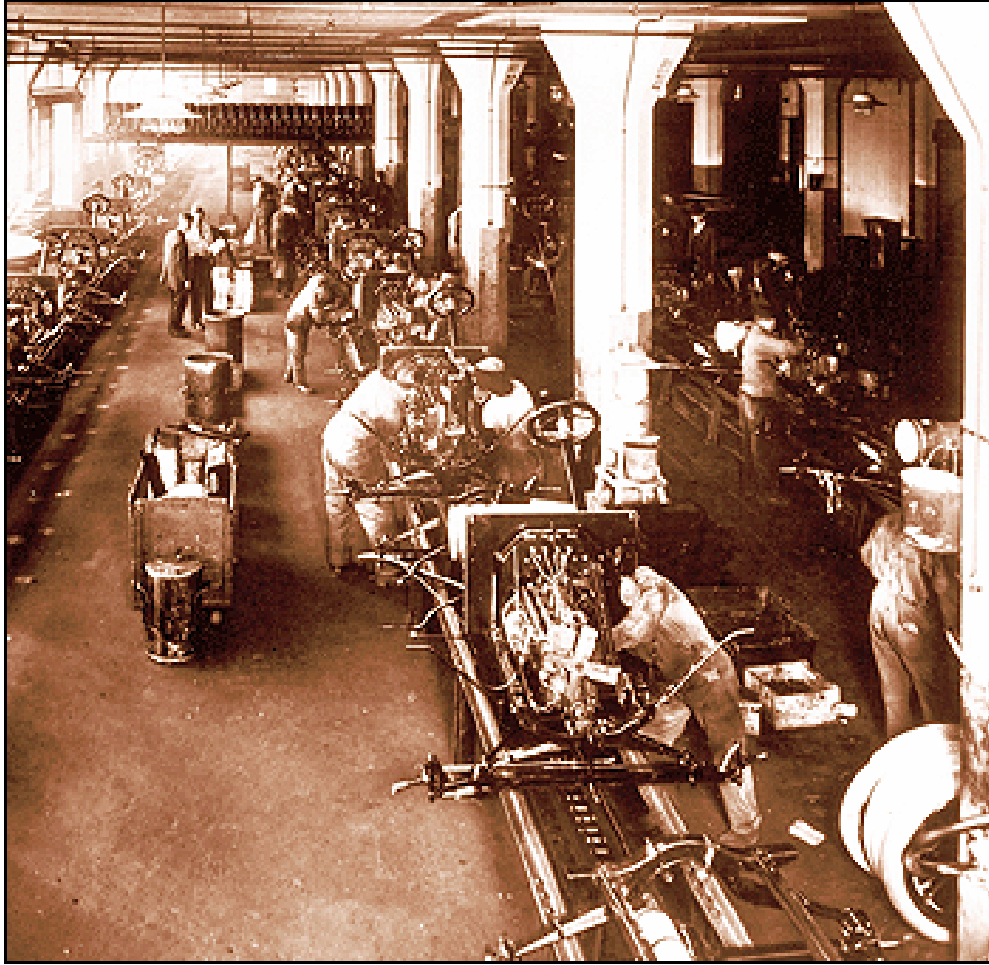
Improving Software Productivity

In hardware design, approaches must evolve in a proper experimental environment in order to scale well. Intel's design and fabrication processes are an example of the evolution of technology evolved in a production environment. We question the ability to achieve software productivity increases that come anywhere near to those of hardware without suitable experimentation in a production environment.

Unlike hardware, software has yet to capitalize on fundamental engineering concepts to improve productivity. This is described by Poore, [73], and also by Anselmo, [2]. Maybe it is time for the software industry to start questioning its own underpinnings and start trying to understand what is required to turn the software productivity curve around.

Part of this effort will require rethinking the programming profession. In the chapters that follow, we will address many issues pertaining to improving productivity in the software industry, and perceptions of the many types of people it employs. Is it really a large union that's just not organized like other unions? Is job security dependent upon ensuring the other guy can't figure out what was done? Or is job security really dependent upon being more productive than programmers in a competitive company, maybe somewhere around the globe?

Survival in a competitive environment depends heavily upon the ability to deal with increasing complexity. This is certainly true for high technology industries, as well as military machines. The organization that is best prepared to deal with increasing complexity has a significant edge. In the long term, those who encumber themselves with unproven beliefs and rituals will be the losers, while those who make determinations based upon scientific findings will be the winners. As history has shown many times over, economics will prevail in the end.



Chapter 2 - Software Productivity

“When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind.....” Lord Kelvin

In the computer field, we have always taken for granted our ability to compare the productivity of one computer versus another. We make decisions to purchase different brands of hardware based upon taking measurements and using benchmarks. Most of us are familiar with MIPS (Millions of Instructions Per Second) and MFLOPS (Millions of Floating point Operations Per Second). Computer speeds determine how fast we get work done. If we have to wait many seconds for the computer to respond to our actions, our work is interrupted. Buying a computer that improves personnel productivity by a factor of 10% can be important to one’s work.

In the early days of computers, people thought of improving their productivity when developing large programs. Various programming languages, e.g., Assembler, FORTRAN, COBOL, etc., were developed specifically to improve programmer productivity. Managers - who came up through the ranks - understood the problems. They compared the time it took to build programs and selected what they considered the best language for their applications. They were able to make informed decisions on what would be used in their shop.

In the 1970s, the concepts of “structured programming” and “top-down design” became important. The literature was filled with approaches to improve productivity in developing software, and most of the contributions made a lot of sense. It was a period of “software enlightenment”. But it is barely remembered today. This is due mainly to the fact that there were no languages that implemented the proposed approaches in a way that ensured improvements in productivity.

Without tools that implemented the concepts of this enlightenment period, it was impossible to measure or realize real productivity improvements. Arguments regarding the desirable features of a programming language appeared to diverge in the literature. Many of the authors were from the academic environment, with no real production experience. Conflicts arose in the literature, conflicts that left open the next big push - to the C language.

C was the result of a small and simple effort at Bell Laboratories to port a game written in FORTRAN. C-like languages were considered a joke by many in the field, and were said to have killed the concept of top-down design. The phrase “write-only language” became popular - implying no one except the original author could read the resulting code.

When building and supporting software, there are many schools of thought regarding the “best” approach. Bookstores are filled with selections on what is best for the programmer. A common excuse for failures is the lack of well stated requirements. Yet the vast majority of programmer hours are spent maintaining existing systems with well-defined requirements. Experienced software managers who came up through the ranks are rare today. When you find them, they generally agree that it is the programming environment that is the major factor affecting productivity. Yet, unlike hardware, there are no accepted measures that afford benchmark comparisons of productivity in building and maintaining software. More importantly, productivity is hardly mentioned in the literature. Comparisons of programming approaches are generally based upon literature advocating a given method. Invariably they lack scientific measures to back up the claims.

What may appear to be unnecessary in solving a classroom problem may be critical in controlling the evolution of a large system in production. Just as with large scale chip design, if an academic institution is not tied into a production environment, it may be difficult for the faculty to understand what may be important to that environment. Unfortunately, many books on software are written by faculty members who rewrite the books they used in school, with little if any real software production experience.

In this chapter we offer a framework that we believe essential to making improvements in software productivity. We start by addressing characteristics affecting the success of a software project.

ISSUES IN SOFTWARE PRODUCTIVITY

Ability To Deal With Increasing Complexity

When building application software with interactive user interfaces, complex databases, dynamic graphics, networks, etc., software complexity grows rapidly. When a large application becomes popular, the user base expands, and functionality requirements can grow well beyond original expectations. As new features are added to a large system, the software becomes even more complex, and the development and support tools are put under great stress, particularly in a production environment. In such an environment, managers are constantly looking at their calendars and wondering when the next slated release will be out. The more facilities contained in that environment to ease the development of new functionality, the higher the productivity.

Scalability

As software product size and complexity increase, the software development environment is stressed in different directions. Various features of a development environment can help or hinder the growth of an evolving product. Scalability is a measure of the size that can be achieved under full control, and the ease with which a software product can continue to grow.

In hardware design, it is well known that approaches not evolved in a production environment typically don't scale well. Intel's approach to chip design and fabrication is an example of the evolution of a good production environment. We question the ability to achieve software productivity increases that come anywhere near to those of hardware without a suitable software environment. That implies relying on a technology that is engineered in a production environment.

Reusability

Reuse is critical to productivity. Reuse is also a major justification for Object-Oriented Programming (OOP). Unfortunately there is no accepted definition of reuse, or a measure of its achievement.

One can take the view that reuse only has meaning when functionality is *inherited* as defined in the OOP sense. Given that most of the functionality already exists in a module, then one must fit the desired functionality and resulting code around the reused module (class) and accommodate differences. We call this "reusability in the OOP sense." Here one must consider the original module, the effort to understand fully the reused module, and the additional code needed to get the full functionality via inheritance. In this approach, one may well inherit functionality that one does not need. Note also that if visibility into what is inherited is low due to hiding, costly conflicts may arise downstream.

The fundamental issue in reusability is the effort required to reuse an existing software module in a new function. We want to minimize the effort (in time and dollars) in support as well as development. This leads to a practical definition of reusability as:

the reduction in effort when one starts with a previous module and modifies it to produce the new function - instead of creating it.

Reusability pays when the amount of modification is small compared to the total effort required to build and support a new module. We must also consider total life cycle costs. Given a development environment that minimizes the life cycle reusability effort, we can expect even higher productivity.

Consider reuse in a production environment. Given that we can copy a module and modify it, the relative amount of change affects our approach. If we must modify a significant percentage of the module, then supporting two distinct modules is hard to argue against. On the other hand, for large complex modules, one may find that the percentage change is quite small. In these cases, the original module is usually composed of sub-modules, most of which remain unchanged. The unchanged sub-modules can become *utilities* that remain intact to support both higher level modules.

But if these modules are hidden, we cannot modify them directly. So it is most important to be able to see these modules and submodules, visually, just as the designer of hardware chips can see the modules. This implies *visualization of the architecture*, a property that has no counterpart in OOP. This has nothing to do with change control, a point of confusion in OOP.

Before addressing measures for comparing software development environments, we must consider measures of the end product in terms of achieving success. Clearly, we must be able to compare what came out of the development environment to determine if it meets the end user requirements. Since we will rarely - if ever - have the luxury to build the same large piece of software using two different environments, we must be able to gauge the relative difficulty in building two different products built in two different environments. The quality of a product, i.e., availability, reliability, and supportability also determines end product success as well. The factors affecting end product success are addressed below.

PROPERTIES OF REQUIREMENTS THAT AFFECT PRODUCTIVITY

As illustrated in Figure 2-1, we are working toward the ability to compare the productivity of different software development environments. Our interest in this section addresses the properties of the product requirements that affect the effort. Clearly, the quality of a product, i.e., availability, reliability, and supportability determines end product success. The properties of a product's requirements that affect the ability to achieve a high level of quality are addressed below. They affect productivity.



Figure 2-1. Measuring productivity of a software development environment.

More importantly, different software development environments will fare differently depending upon these properties. Small classroom exercises can be produced quickly in simple environments. But these environments may fair poorly when developing and supporting large complex products.

Functionality

Poorly specified requirements are often cited as the cause for late and buggy software. Sometimes this is true. However, the authors are aware of multiple cases where functionality was well specified, including user-supplied test data to determine whether requirements were met, and the software efforts still failed. In fact, the vast majority of software man-hours are spent in the support mode where the requirements are generally well known, and the productivity is considered low.

A more important factor appears to be the amount of functionality one must deal with. We must be able to quantify the size and complexity of the function space specified for a software product in order to determine the difficulty one faces in development and support for that product. This has been addressed in the function-point method, see Capers Jones, [52], and [53].

Additionally, successful software systems typically serve an ever-widening range of functionality. When comparing software development environments, one must evaluate their ability to handle the increasing functionality of a software product as it evolves in the marketplace, pushing the need for scalability in the development environment.

Complexity

Having good descriptions of all of the functions to be built into a software product is not likely to be sufficient when trying to predict the level of difficulty to produce it. The level of complexity of each function must be considered as well. Productivity can take on significant variations due to different levels of complexity of implementation of the functions. Complexity factors can be categorized for different functions so that a weighted measure can be derived. But, they are hard to predict.

The difficulty in assessing complexity is particularly true when developing complex algorithms with huge decision spaces. Often, one does not know all the cases to be dealt with until one is well into testing. Having an environment that supports the growth of complex algorithms, as they are expanded to handle all of the unanticipated cases, can help to improve productivity dramatically. We also note that an environment that provides the ability to easily isolate and test software modules also improves productivity.

Quality

As functionality and complexity grow, the number of opportunities for bugs multiplies. Knowing that two pieces of software have equal numbers of new bugs found per month is not sufficient to determine the comparative quality of each. One may have much more functionality than the other. Furthermore, many more people may be using one, and using it more heavily, than the other. These factors must be accounted for when comparing the quality of different pieces of software.

Quality of software can be measured in terms of the availability of its specified functions, and the time and cost to support that software to maintain an acceptable level of availability. The acceptable level of availability will be determined by the users of that software, particularly if they have a choice. Measures of availability can incorporate the level-of-usage factors for all functions. In the following we assume that software is designed to meet a quantified level of quality, as described in [23], and [1].

LIFECYCLE CONSIDERATIONS

Figure 2-2 depicts two characteristics that can be used to understand productivity of a software development environment. The top characteristic shows an investment curve for development and support of software. The area under the curve represents the product of time and cost per unit time, yielding the total dollar investment to build and support a piece of software. For software products, more time and money is spent supporting product enhancements and error corrections than in original development.

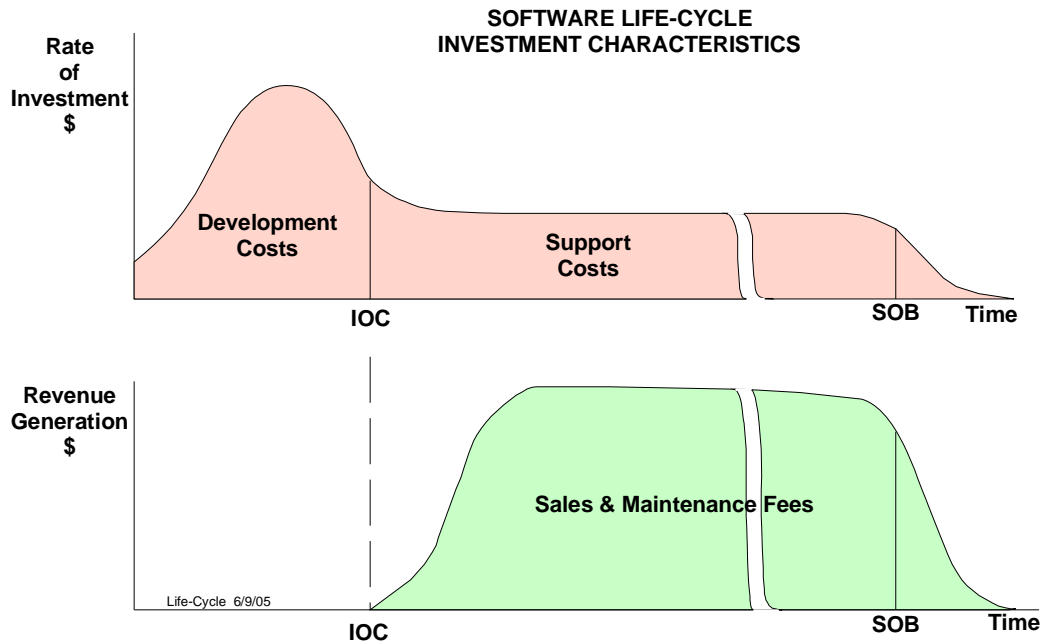


Figure 2-2. Measuring the productivity of software development and support.

The second characteristic illustrates the revenues generated from product sales and maintenance fees per unit time. Revenues start to flow when an Initial Operational Capability (IOC) is reached, and start to cease upon System Obsolescence (SOB).

If the development time (to IOC) is stretched out, and total development costs remain constant, i.e., the expenditure rate is slower, then the time to reach revenue growth is pushed out. Total revenues are reduced if competition gets in earlier, or if the product becomes obsolete. This causes loss of Return On Investment ($ROI = \text{total revenue} - \text{total investment}$). This can happen if initial product quality is not sufficiently high, since customer dissatisfaction will inhibit sales growth and encourage competition.

Improvements in productivity must be reflected in improvements in ROI. Therefore, productivity is inversely proportional to the costs incurred. This comprises development costs and support costs. Additionally, if development costs remain fixed while IOC is reached in half the time with equal quality, revenues can start flowing earlier. This implies that if developer A spends money twice as fast as developer B, but reaches the same quality at IOC in half the time, A can expect a higher ROI. It takes much higher productivity for A to achieve this.

The total cost, C, can be estimated as

$$C = K \cdot M,$$

where M is the total man hours expended during development and integration, and K is a loaded man-hour cost that depends on overhead, general and administrative expenses. We note that this only reflects the cost part of productivity. As indicated above, a highly productive developer will benefit from completing a project in half the time, even though the total cost may be the same. Thus, if the length of time to complete the project is factored in directly, then productivity may be inversely proportional to the total cost multiplied by the project duration, T. This is factored into the prior estimate.

$$C \cdot T = K \cdot M \cdot T$$

We are not stating that this is *the* measure of (inverse) productivity. We are asserting that one must conduct experiments and take measurements to validate such an hypothesis. We encourage other hypothesis; but whatever the measure, it must be backed up by a scientific method, using valid repeatable experiments.

We note that the support mode is typically dominated by incremental developments (enhancements), and can be treated accordingly. We also note that, if a given level of quality is achieved for competing software systems, then the revenue side is accounted for fairly, since other factors, e.g., marketing costs, are neutralized.

DESIGN & IMPLEMENTATION PROPERTIES AFFECTING PRODUCTIVITY

Having addressed the important external (product requirements) variables that affect productivity, we can now investigate the internal product design and implementation properties that affect productivity. Our goal is to characterize a software development environment that, based upon our experience, supports these properties so as to reduce the time and man hours to develop and support a software product. Ideally, we would like to identify a minimal set of orthogonal factors. To this end, we offer the following factors.

Independence

When attempting to reuse a module, one must be concerned with the independence of that module relative to its use by other modules. If the reused module is not in the same task, then one may have to copy it, e.g., as a library module, for use in different directories or platforms. If it needs other modules to operate, they also must be copied.

The more a module is tied to (i.e., shares data with) other modules in a system, the higher its *connectivity* to other parts of a system. The connectivity (number of connections) is measurable. The higher the connectivity, the lower the independence. When designing hardware modules to be independent, one works to reduce the number of connections to other modules to a minimum. We note that visualization of the architecture is critical to performing this design task for hardware. This is true for software as well.

When building software using OOP, class abstractions cloud the ability to visualize connections. Understanding how data is shared between software modules can be difficult, especially when inheriting classes that inherit other classes. It is hard to simply “inspect” a module to determine its degree of connectivity and understand the way it interacts with other parts of the system.

Hiding and abstraction in the OOP sense make it difficult to pull (copy) a module from one system and place (reuse) it in another. This difficulty in reuse, from a production standpoint, stems from the effort required to measure the level of independence in a typical OOP environment. More importantly, if one cannot measure it, one cannot design for it, let alone improve it.

In the case of hardware, one designs for minimum connections between modules. One uses CAD tools that provide a visualization of the architecture to do this. Connectivity (coupling) is a key property affecting design productivity. This is true in software as well. But to fully understand this principal, one must be able to “see the architecture” and inspect the connections *visually*.

Understandability

When managing a large software project, one gets to witness the loss of productivity that occurs as two programmers reinvent the same module. Productivity is lost trying to decrypt algorithms and data structures that are coded so as to minimize the number of keystrokes used to write them, or to maximize “economy of expression”.

If these algorithms are passed on to someone else, they may become enveloped in comments to explain the code, sometimes multiplying the size of a listing by whole numbers. Some claim that understandability of a language can be gauged by the average number of comments in well documented code. Taking time to choose good names - and minimizing their reuse for different purposes - is paid back many-fold in a large system. In our view, understanding the code *directly* is a major factor in productivity of software, especially in the support phase of the life cycle of a product. The improvement in using understandable notations has been measured by Ledgard, [59].

More important than names is the use of control structures. This has been emphasized by many previous authors, particularly Mills, [66]. This is a significant property affecting productivity when building complex algorithms. This is addressed further in Chapter 12.

More important than language is the underlying architecture of a system. This property is hard to envision if you have never seen a direct visualization of software architecture. This is only accomplished if there is a one-to-one mapping from drawings of the architecture to the physical layer, i.e., the code, just as there is in a drawing of a complex computer chip. We believe that without this visualization, significant productivity improvements can never be achieved for software.

Using OOP, the opposite situation occurs. The architecture - if it exists at all - is hidden behind the code - the only representation of the real system. Diagrammatic representations are abstractions that do not reveal the true complexity or hidden dependencies.

Understandability of the architecture contributes directly to the design of independent modules. We believe that one can measure the visibility of software architectures as provided by different development environments and relate that to productivity.

Flexibility

One motivation behind the Extreme Programming movement, as well as Microsoft's software development philosophy, is the incremental approach to software. This was the topic of a book by Cave in 1982, [24]. In this approach, functionality can be added in small pieces, often with a working "daily build". This requires an environment that supports this approach.

Computer-Aided Design (CAD) tools make hardware architectural changes easy, especially when a system has been designed on a modular basis. A CAD system that does the same for software, i.e., starts with a visualization of the architecture on a modular basis, and provides a one-to-one mapping into the detailed code, can ensure design independence of modules while allowing visibility of the desired details. This capability in hardware engineering is alluded to in Poore's paper, [73]. Based upon first hand experience, we can attest that this capability, embedded in a software development environment, provides real reusability.

With such a flexible facility, one can design a little, build a little, and test a little, growing a system incrementally to ensure components are meeting specifications and showing near term results. One can quickly detect when changes cause components to fall out of specification ranges. Fault isolation is much more easily accommodated. These factors all lead to higher productivity.

Visibility

Electronic circuits are described by systems of differential equations. Yet, it is hard to imagine designers working without drawings of circuits. As circuits get large, e.g., thousands of elements, it is the visualization of the architecture - the parsing of functions into iconic modules and lines to show how they are interconnected - that becomes overwhelmingly important. Visualization of the architecture is the key to productivity.

We claim this is also true with software. However, one must achieve a one-to-one mapping from the architecture diagram to the code in order to gain the benefits derived from the equivalent in hardware. This is only achievable when data is separated from instructions as described by Cave, [25]. If there is a silver bullet in software, this is it. Productivity gains can multiply using this CAD technology so as to achieve the equivalent of a Moore's curve for software allowing large complexity increases every year.

Abstraction

No one can argue the usefulness of abstraction. It certainly can help to get through major design problems. It can also serve to sever ties to reality in a production environment. It is easy to draw block diagrams for sequential tasks that relate to the code. But in highly interactive systems, mouse and keyboard event handlers support many functions, and the software architecture becomes orthogonal to user functionality. Block diagrams lose meaning when one looks at a software design from the extremities of the functional interface to the detailed code that manages databases and devices.

PRODUCTIVITY OF SOFTWARE DEVELOPMENT ENVIRONMENTS

We can now address the properties of a software development environment that lead to higher productivity. Simply put, it is an environment that *best supports the productivity properties of the requirements, the life cycle, and the architecture and implementation of the desired final product*. The properties described above can be used as proxies to measure the development environment. For example, how easy is it for a newcomer to a project to understand the architecture, or the code? How easy is it for that person to reuse already developed modules, possibly modifying parts of them to suit different functionality? How easy is it for someone to take over a large set of modules without the original author? Just as in hardware, these properties can be observed directly by knowledgeable managers in a software production environment.

CONDUCTING EXPERIMENTS TO MEASURE PRODUCTIVITY

Borrowing from DeMarco's *Controlling Software Projects*, [35], "You can't control what you can't measure." Before we can expect to improve productivity, we must measure it.

Since we will rarely - if ever - have the luxury to build the same large piece of software using two different environments, we must be able to gauge the relative difficulty in building two different products built in two different environments.

Apparently, benchmark comparisons of different approaches to developing software do not exist because of the size of experiments envisioned to perform the task. People envision two teams developing a sufficiently complex piece of software using competing environments. One can see why such an expensive undertaking is not done.

But most experiments in science are not very large in scope. Focus is usually on creating sequences of small experiments that can lead to larger conclusions. We believe software can be broken into pieces such that the methods that produce them, including integration, can be examined experimentally.

But just as computer chip manufacturers are constantly taking data to improve productivity, both in the design phase and the production phase, so can software product companies. Managers in competitive environments are always looking to cut costs and time while improving quality. This implies that management understands the details sufficiently to guide change, and that the designers and programmers are motivated to be on the same sheet of music - trying to improve productivity while maintaining - if not improving - quality. Unfortunately, there are many environments in which neither case exists.

CONCLUSIONS ON PRODUCTIVITY

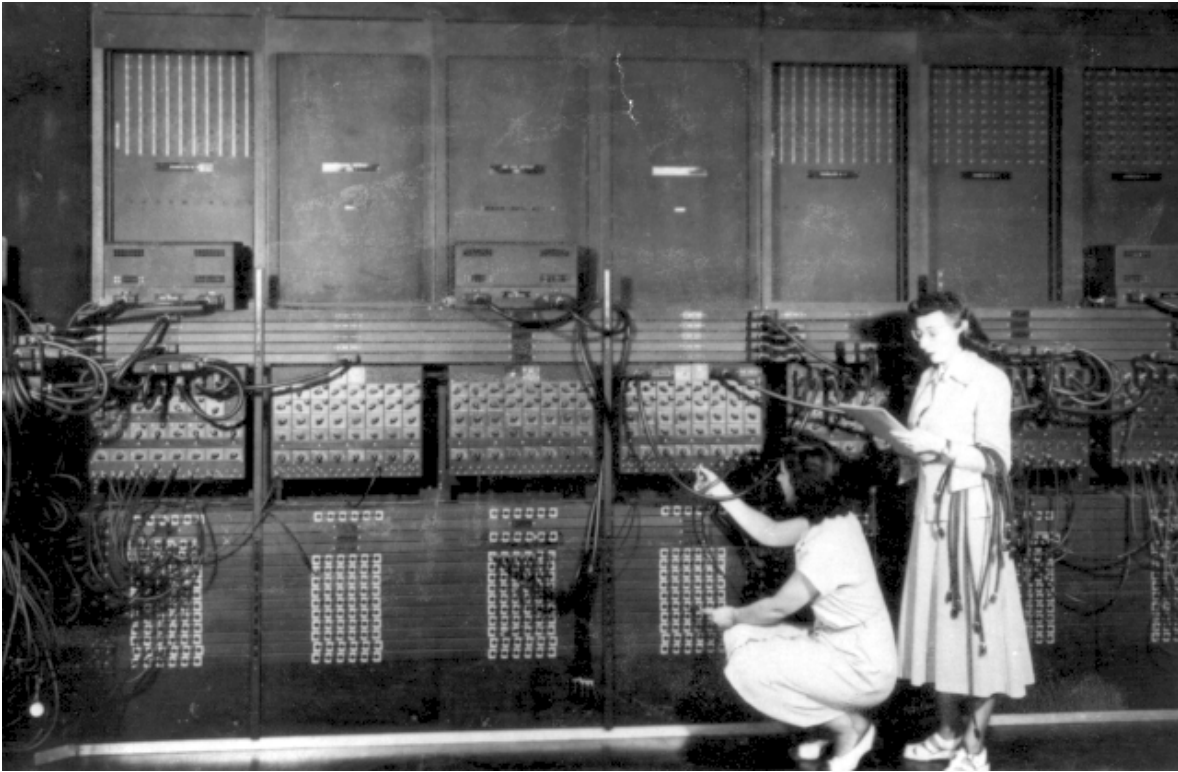
As the quote from Kelvin implies, we cannot expect to improve software productivity without measuring it. The measures of a software end product - functionality, complexity, and quality - are not new. They form the foundation for measuring productivity.

If a given level of quality is achieved for the same software system by competing organizations, their relative productivities may be measured as being inversely proportional to the product of their cost and development time.

Productivity of a software environment depends upon the understandability and independence of modules produced. These are inherent properties of a software system that can be increased or decreased by design. Development environments that provide good visualization and CAD techniques can help software designers to construct systems with these properties just as they do for hardware designers.

Finally, we must be able to measure changes in productivity to validate our assumptions regarding its dependence on these properties. We believe that this can be done using a large number of small experiments that, combined statistically, will represent the productivity of a development environment. We perceive that such experiments are suitable for university collaboration.

In this book, we hope to make it clear. It is time for the software industry to start questioning its own underpinnings. It is time to start seeking the truth and trying to understand what is required to turn the software productivity curve upward. Part of this effort will require rethinking the programming profession. Is it really a large union that's just not organized explicitly like other unions? Is job security dependent upon complex code that another programmer can't figure out, therefore making the first indispensable? See [94]. Or is job security really dependent upon being sufficiently more productive than programmers in a competitive company, maybe somewhere around the globe? If you think the answer is the latter, read on and consider how you can help to make the software industry more productive.



Chapter 3 - A HISTORY OF PROGRAMMING **- LESSONS FROM THE EARLY DAYS**

Authors' note: *In presenting the history below, we emphasize two important ideas in software. First, “independence”, by which we mean the ability to make modifications to software without affecting other portions of the software. And second, “understandability”, by which we mean the ease with which a programmer can read (and thus change) a portion of the software written by another.*

IN THE BEGINNING ... (circa 1955 - 1965) - *DRAMATIC JUMPS!*

Back in the old days we wrote code in ones and zeros. To be a programmer, one had to understand the machine. Programming involved registers, arithmetic instructions, control instructions, I/O instructions, the program counter, etc. Even writing a simple program was not easy. One had to define what was in each memory location one expected to use. Figure 3-1 illustrates the format of a program. This example is for a fictitious, but simple, single address machine with an A register. For example, OP Code 1000 cleared the A register to zero and then added the contents of the specified memory address, e.g., [13], into A (in our example, location 13 contains the value 25).

Notice “the separation of data from instructions”. The data could be put anywhere. The instructions had to follow in sequence, unless a transfer (GOTO) was used.

MEM LOC	OP CODE	MEMORY ADDRESS	COMMENTS
1	1000	00001101	CLEAR AND ADD [13] TO A
2	1001	00001110	ADD [14] TO A
3	0010	00001111	READ TAPE INTO 15
4	1010	00010000	SUBTRACT [16] FROM A
5	0111	00001110	STORE A IN 14
6	1110	00001011	TRANSFER TO 11 IF A IS NEGATIVE
7	1100	00000010	TRANSFER TO 2 IF A IS POSITIVE
8	1001	00001111	ADD [15] TO A
9	0011	00000000	PRINT A
10	1111		STOP
11	0011	00001111	PRINT [15]
12	1111		STOP
13	00011001		25
14	00011010		26
15	00000000		0
16	00110010		50

Figure 3-1. A computer program written in binary.

Interestingly enough, working in the binary number system was not the difficult problem. The problem was changing the program. Let's suppose we wanted to put a few additional instructions into this program starting at memory location 8. Then, all entries from there down get new memory addresses, implying that every reference to them must be changed, a real mess! Even if we were clever enough to insert a GOTO to some higher location with our new instructions, we still had to move the old instruction in 8 to the new location, and put another GOTO at the end of the new sequence (to get back). Time spent debugging these changes and random jumps was immense. The important lesson here is that all lines of code were dependent upon the sequence, and thus each other. *This lack of independence made change very difficult.*

The Properties of Understandability and Independence

It didn't take much time for people to start writing translators to make programs more readable, and therefore more *understandable*. The first simplification was the use of mnemonic names for Op Codes and using decimal numbers for addresses instead of binary.

A major step toward improving productivity was the *assembler*. Each machine had its own assembler, written by the manufacturer. The first assemblers allowed programmers to eliminate the use of actual numeric addresses for data as well as transfer points. Instead, alphanumeric labels were allowed. Because these reference identifiers were no longer sequential, and were totally *independent of position*, one could insert new instructions anywhere without disturbing the rest of the code. Figure 3-2 illustrates a simple program written in an assembly type language.

LABEL	OP CODE	MEMORY ADDRESS	COMMENTS
RESTART	CLA	X1	A = X1
	ADD	X2	A = A + X2
	RDT	Y1	READ TAPE INTO Y1
	SUB	Y2	A = A - Y2
	STO	X2	X2 = A
	TRN	END2	TRANSFER TO END2 IF A IS NEGATIVE
	TRU	RESTART	TRANSFER TO RESTART IF A IS POSITIVE
	ADD	Y1	A = A + Y1
	PRN	A	PRINT A
	STP		STOP
END2	PRN	Y1	PRINT Y1
	STP		STOP
X1	25		
X2	26		
Y1	0		
Y2	50		

Figure 3-2. A computer program written in assembly language.

As the desire for new functions and features expanded, and more memory became available, computer programs started to grow in size. Drum memories became sufficiently reliable so that sections of larger programs could be stored on the drum, and rapidly overlaid into main memory when they were needed. The concept of standard overlay modules soon became a necessity, particularly for handling I/O devices such as keyboard input, printed output, paper tape input and output, etc. This led to the separation of programs into subroutines, with provisions in the assembler language for jumping to them by name.

As overlays became popular, one had difficulty laying out the patchwork of where routines would be mapped into main memory. This was solved using relative addressing at the subroutine level. Assemblers that translated code on a relative address basis were known as relocatable assemblers, implying that the actual starting addresses of the program counter and data blocks remained undecided until they were loaded. It was still up to the programmer to do the overlay and set the starting address pointers. But, this provided for *spatial independence* of overlays, relative to where they were mapped into main memory, making them much more reusable.

Independently Linked Subroutines

And so, as subroutines became much more reusable, programs grew even larger. This led to another problem. All routines belonging to an overlay had to go through the assembler in one shot. As main memory became larger, making a simple change to a single routine still required that the whole overlay be reassembled.

This problem led to the development of a separate link and load phase, wherein a single subroutine could be assembled *independently* of the rest. This subroutine could then be relinked with the rest of the overlay that was previously assembled into a partially linked object module. As a final step, a load module was produced with the absolute addresses resolved. The software that provided this facility was called a linking loader. This allowed subroutines to be built and assembled independently, making them and their object modules the basic reusable elements. Object modules resided as independent entities in library pools that were scanned during the link process.

A growing list of library routines created the next problem, that of duplicate names. To this day, the problem of duplicate object module names, amplified by flat file object libraries and very simple library managers and linkers, plagues a growing part of the programming world. This has led to a lot of bandaids in programming languages to cover up problems that are properly solved at the environment level. We will address these problems downstream.

Flow Charts

Because of the difficulty in understanding assembler code, and particularly the instructions for transferring control, programmers created the *flow chart*. When using boxes and diamonds to illustrate functions and decisions, a symbol on the flow chart typically encompassed many instructions. So the number of lines of code was larger than the number of flow chart symbols. These advantages disappeared with understandable languages and control constructs.

THE FIRST BIG JUMP IN UNDERSTANDABILITY - FORTRAN

The desire to make the programming job easier led to still another major step toward making the machine do more of the work of understanding what the human meant. People writing programs to solve large sets of mathematical equations were the first to invent a more *understandable language* and corresponding translator - the FORMula TRANslator (FORTRAN). FORTRAN shifted the burden of translation from a language that humans could easily read and write, onto the computer. Productivity went way up because of a number of factors.

- One person could understand much more easily what another person wrote. This allowed a large program to be constructed with a team effort. It also allowed completion of an effort and reuse of code without the original author.
- Many errors endemic to assembly language disappeared. Probably the most common was scribbling on instructions (and immediately the rest of memory.) FORTRAN took away the Von Neumann facility of being able to write instructions that modified themselves (some assemblers prohibited this also).

- More and more smarts were built into the translation process as people learned what it took to be more productive. These included improved syntax, various forms of error checking and prevention, run time messages, etc.

It is interesting to note that many programmers of the day looked askance at FORTRAN, disagreeing with the above bullets for various "technical" reasons. One of these was efficiency of the code produced, until it was recognized that it was a rare programmer who could do as well as the designers of automatic code generators. In spite of this resistance, FORTRAN became one of the best examples of the following:

When understandability takes a leap, so does ease of change, and thus productivity.

Anyone racing to build computer programs to solve mathematical problems quickly got on board the FORTRAN train. If they didn't, they couldn't compete.

For people building data processing systems, FORTRAN left a lot to be desired. It was cumbersome to work with files having complicated record structures. The FORTRAN FORMAT statement is a quick way to get listings of columns of numbers, and some alphanumeric data, but there was no friendly mechanism for creating the complex data structures necessary for dealing with large data files. Even the data structure capabilities existing in advanced versions of FORTRAN today leave much to be desired.

A major problem with FORTRAN is the fall through approach to coding that is a carry over from assembly language coding. Every line depends upon where it falls in the sequence. Labels exist for looping and GOTOs but, in general, one cannot isolate blocks of code inside a subroutine and move them around without great difficulty. An example of a very efficient sorting algorithm, published in the ACM Journal in 1969, [93], is shown in Figure 3-3. This algorithm is very efficient at sorting, and is a clever algorithm with a sophisticated mathematical background. Unless one is familiar with the implicit statistical methods for sorting referenced in the paper, one is hard pressed to understand the underlying algorithm. The example is not meant to reflect poorly on the excellent work of the author. Rather it is a reflection on programming style and practices in that era. Note that, to save time, GOTO's are used to replace DO loops. This accentuates the fall through approach. As an exercise, try putting this example into a flow chart.

This program also exemplifies "economy of expression." A minimum number of keystrokes is required to retype it from the journal - an important consideration of the time. One can also imagine being assigned to make changes to a five to ten page subroutine of this nature - clearly a humbling experience for a rookie. Of course, things just aren't done that way anymore (we hope), at least not in FORTRAN. We strongly suggest that economy of expression is *inversely* correlated with the overall life cycle economics of a large software product. We believe that this can be verified by experimental evidence, e.g., that reported by Fitsimmons and Love, [38], Ledgard et al, [59], and Sitner, [92].

```

SUBROUTINE SORT(A,II,JJ)
C SORTS ARRAY A INTO INCREASING ORDER, FROM A(II) TO A(JJ)
C ARRAYS IU(K) AND IL(K) PERMIT SORTING UP TO 2**(K+1)-1 ELEMENTS
  DIMENSION A(1), IU(16), IL(16)
  INTEGER A, T, TT
  M=1
  I=II
  J=JJ
  5 IF(I .GE. J) GO TO 70
  10 K=I
  IJ=(J+I)/2
  T=A(IJ)
  IF(A(I) .LE. T) GO TO 20
  A(IJ)=A(I)
  A(I)=T
  T=A(IJ)
  20 L=J
  IF(A(J) .GE. T) GO TO 40
  A(IJ)=A(J)
  A(J)=T
  T=A(IJ)
  IF(A(I) .LE. T) GO TO 40
  A(IJ)=A(I)
  A(I)=T
  T=A(IJ)
  GO TO 40
  30 A(L)=A(K)
  A(K)=TT
  40 L=L-1
  IF(A(L) .GT. T) GO TO 40
  IT=A(L)
  50 K=K+1
  IF(A(K) .LT. T) GO TO 50
  IF(K .LE. L) GO TO 30
  IF(L-I .LE. J-K) GO TO 60
  IL(M)=I
  IU(M)=L
  I=K
  M=M+1
  GO TO 80
  60 IL(M)=K
  IU(M)=J
  J=L
  M=M+1
  GO TO 80
  70 M=M-1
  IF(M .EQ. 0) RETURN
  I=IL(M)
  J=IU(M)
  80 IF(J-I .GE. 11) GO TO 10
  IF(I .EQ. II) GO TO 5
  I=I-1
  90 I=I+1
  IF(I .EQ. J) GO TO 70
  T=A(I+1)
  IF(A(I) .LE. T) GO TO 90
  K=I
  100 A(K+1)=A(K)
  K=K-1
  IF(T .LT. A(K)) GO TO 100
  A(K+1)=T
  GO TO 90
  END

```

Figure 3-3. Example FORTRAN program published in the late 60's.

Although FORTRAN has come a long way since it was first offered, many problems still exist, causing it to be used less and less each year. The problems described above caused the desire for a new approach early on, particularly for large data processing programs, and a new language was produced in the early 60's to fit the bill. This was COBOL.

THE SECOND BIG JUMP IN UNDERSTANDABILITY - COBOL

The COmmon Business Oriented Language (COBOL) was developed by experienced programmers to achieve a common goal - build a language that humans could easily understand, one that could read close to plain English. To the extent that COBOL quickly became owner of approximately 80% of the world's code for about two decades, it was the most successful programming language ever devised. An October '95 article in Inform, [51], cites studies by IDC, Gartner, and Dataquest that showed COBOL still accounted for over 53% of all applications in the world, 80% of all business applications, 50% of all new business applications, and 5 billion lines of new code added each year. This is because of its ability to improve real economic measures of programmer productivity, where it counts - in the maintenance phase of a life cycle. And these improvements are clearly due to its *understandability*.

Yet, no language has been more maligned by a vocal segment of the software industry than COBOL. And, this is not a new phenomena. In the 1960's, when the financial industry in New York City was going through conversions to new IBM-360s, costs to upgrade software were going through the roof. At that time, experienced programmers insisted that accounting applications could only be written efficiently in assembly language. (Neither EXCEL nor LOTUS existed then.) What they were really concerned about were armies of high school graduates that were marching into Manhattan and dramatically lowering the cost of building new software using COBOL.

Data processing managers had to fight to dislodge the company software assets from the hands of the assembly language programmers and turn them over to a younger, less skilled workforce who could write code that everyone could understand. In that highly competitive economic environment, it was only a matter of time. The cost of software development and support plummeted with COBOL, and the leftover money was spent developing more sophisticated applications. COBOL also created a separation of skills, and a separate workforce of systems programmers still using assembler and Job Control Language (JCL).

As scientists, we cannot ignore the success of COBOL. We must understand the facts behind its ability to cut costs and improve productivity. Certainly, one cannot contest the readability of COBOL relative to any other language. Greater readability leads directly to understandability. Next, COBOL implemented the one-in one-out control structure advocated years later by Mills, [66]. The objective of this control structure is to eliminate "waterfall" or "fall through" coding, providing a hierarchy of blocks of instructions, within a subroutine. This additional layer of hierarchical structure can serve to increase the understandability of subroutines, a feature that does not really exist in other languages. However, as we will describe below, the COBOL implementation hindered the desired improvements in logical clarity.

COBOL's ability to process data has been unsurpassed. The most important factor in data handling is the data description. COBOL allows a user to organize data structures the way one wants to see it, hierarchically, by logical organization. Not by type. Furthermore, What-You-See-Is-What-You-Get (WYSIWYG) in memory. There is no such thing as "word boundary alignment" behind the scenes. Most programmers do not understand the importance of this feature unless they have done sufficient character string manipulation or data processing using a character oriented language. If one has never had this feature, one can't appreciate how great it is! It's what allows one to do group or subgroup moves from one data structure into another, e.g., moving part of a message or record, defined as all character data, into a template defining each field. It provides for redefinition of data areas so that they can be looked at using different templates or filters without moving the data.

One cannot do these things in any language that permits word boundary alignment to go on behind the scenes. Until VSE, described in Section 2, no language has provided these data structure facilities nearly as well as COBOL. And these are critical when dealing with complex data structures.

Caveat - The Very Large Subprogram Problem

We must also understand the weak points of COBOL. One of these is breaking large programs into subprograms. This is a result of COBOL's heritage of sequential batch oriented jobs. This is a difficulty for COBOL. Although COBOL provides a subprogram capability, it is not easily used. This has led to extremely large COBOL programs that are difficult to change and maintain.

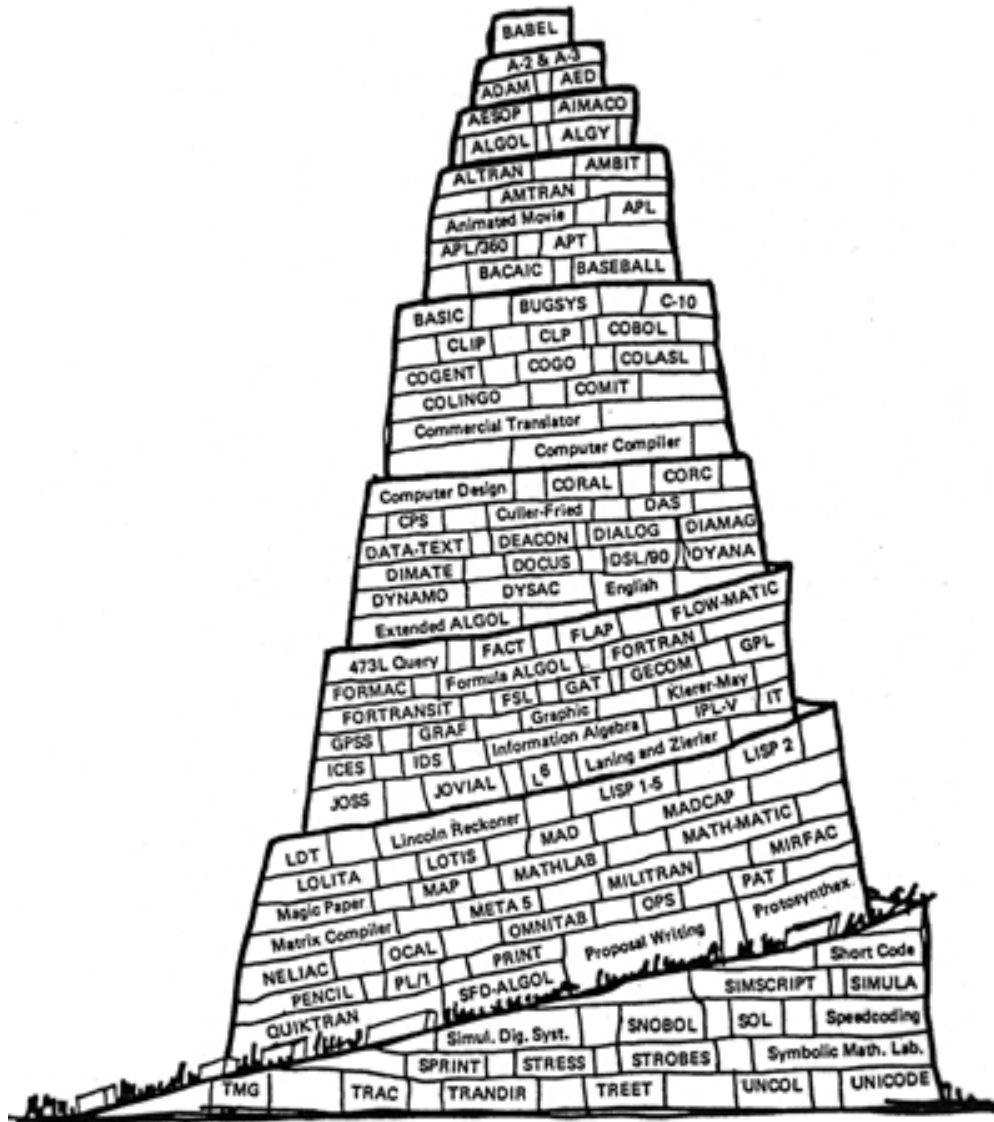
The reason that COBOL programs are not easily broken into subprograms is subtle. COBOL's sharing of data structures between subprograms by pointer is clearly superior to passing individual data elements. However, the mechanism for accessing data structures poses a problem since each structure must be declared in "working storage" before it can be used by another subprogram, where it then must be declared in a "linkage section." These two classes of declaration impose a constraint that makes it difficult to structure, and especially to restructure, an architecture. It is amplified by the requirement that, in a calling chain, if any routine down the chain wants access to the structure, it must be declared in all routines along the way. One cannot switch or discard the "MAIN" routine without a big upheaval.

As indicated above, COBOL contains a one-in one-out control structure as advocated by Mills. However, the implementation via the PERFORM *paragraph* statement does not preclude the waterfall from one COBOL paragraph to the next, hindering the ability to achieve the desired level of logical clarity. Another implementation "feature" allows programmers to PERFORM sequences of paragraphs, further maligning potential clarity. These sequences become especially difficult to follow when they are exited by GOTO statements that can jump control anywhere, including the middle of another sequence somewhere else in a large subprogram. This problem seems to be exacerbated by COBOL's unusually large subprograms.

The Tower Of Babel - Programming Languages

Although we have only discussed FORTRAN and COBOL, many other early languages had their impact on the software development process. Although some of these languages have had substantial followings during certain time periods, none have matched the long-term success of FORTRAN and COBOL. ALGOL was developed in the early 1960s, partly as an algorithm specification language, one that could be used to specify the details of computer architectures. It was the language used for papers in the Association of Computing Machinery (ACM) Journal. It was the principal language of the Burroughs 5500, one of the earliest time-sharing machines.

SIMULA was another early language, used for simulation. Although hardly used in the U.S., it is referenced frequently. PL/1 was IBM's answer to provide one language to take the place of COBOL and FORTRAN, a noble goal. However, it was never close to COBOL from a readability standpoint, and had so many options that programs were very difficult to understand and debug. APL is an excellent language for solving vector - matrix equations, but is scientifically oriented. PASCAL and BASIC have been well utilized, but have never reached the acceptance level of COBOL or FORTRAN. We will simply mention that each of the U.S. Department of Defense services invented its own language: TACPOL (the Army), CMS2 (the Navy), and JOVIAL (the Air Force). Each language was "justified" based upon the unique requirements of its particular military environment. That is, until Ada came along and the U.S. Department of Defense mandated the use of Ada to replace them all. But, it did not get as far as PL/1.



Chapter 4 – A More Recent History

“Continued and rapid growth in the power of hardware has not only enabled new applications and capabilities, but has permitted sloppy, unprofessional programming to become the virtual standard of business and industry. Hardware has allowed the software profession to avoid growing up, to remain in an irresponsible adolescence in which unstable products with hundreds of thousands of bugs are shipped and sold en masse.” -- Larry Constantine [30]

THE WAVES OF A SOFTWARE REVOLUTION

Revolutions impose sweeping changes upon the frame of reference people use to value a system. They can impart significant improvements, wreak havoc, or anything in between. Major revolutions are usually preceded by smaller waves, ones with a smaller force. For software, the major wave has not yet come - but it is on the horizon. In articles written by prominent people in the software field, e.g., [1], [58], [61], [73], there is a sense that people are getting ready to accept a major paradigm shift. Hardly anyone at a managerial level is disputing the need for it. To help gain an understanding of what the needs are, we will take a look at more recent history to consider the prior waves - the previous attempts at a software revolution. Our main purpose in this chapter is to prevent this history from repeating.

THE STRUCTURED PROGRAMMING WAVE (circa 1970 - 1980)

The first wave of a software revolution came in the 1970s. It was a noble effort, a period of enlightenment. Serious thought was put into principles for cutting the growth in project failures and costs of software. The literature expanded rapidly, with contributions derived from case histories. The “Mathematical Foundations of Structured Programming” by Harlan Mills, [66], was considered a major contribution. It describes the technical properties of one-in/one-out control structures, and their corresponding understandability. This was one of the first papers that tried to set the everyday programming problems in a framework for good scientific analysis. Academic inputs came from many contributors, e.g., Dijkstra, on GOTOless programming. Dijkstra also provided a more mathematical direction for programming improvement.

But there were clearly great disparities in the productivity of software organizations at the time. The high correlation in disparities became most apparent when software organizations were grouped into government versus their commercial counterparts, with the government lagging far behind in most areas. Exploding budgets and time schedules were absorbed by taxpayer dollars without knowledgeable oversight. Much of the literature pointed out the problems of poor management and provided guidelines, procedures, and standards to insure control of the lifecycle, e.g. [23]. Other software projects simply grew because nothing was coming out, and the end justified the means. Much of this was due to an even greater disparity: that of understanding the problems of developing anything complex, not just software.

Much of the problem stemmed from the lack of good operating systems. These complex systems were just evolving. IBM was hit by this phenomenon, particularly on the OS/360 project, and much history was analyzed and published. This included the excellent combination of technical and management principles offered by Sherr, in **Program Test Methods**, [90], who defined the basic regression test method as it applied to software. Another describes the direct experience of Fred Brooks, the principal architect of OS/360, published in his classic *Mythical Man Month*, [12], a best seller. Another major contribution was Baker's Chief Programmer Teams, [5], one of the original publications on top-down design and organization of the programming staff.

A large number of publications expanded upon these ideas, or brought forth excellent new ones. From the literature, it appeared as though the world of software was going to change forever in these new directions. But most of it has been forgotten. Most computer science graduates don't know about Harlan Mills. The literature from the 1970s is hardly referenced (unless it refers to C and UNIX). The ideas were excellent and people accepted them. But, when they turned around and went back to work, they had little to use to implement the principles. This is because no software environment existed to support the most important concepts.

The lack of an existing environment cannot be considered the sole reason for lack of success of this movement. It was also due in part by the IBM PC, where most of the original software had to be written in BASIC. Some software houses today still rely on BASIC programmers. A more long term influence was the relatively huge funding (\$Billions) for UNIX and C by AT&T, and the U.S. and other governments. And people go where the money is. We now cover the period of falling productivity - the dark ages of the software field.

THE RISE OF UNIX AND C (circa 1975 - 1985)

The "Paper Tiger" Revolution of the '70s gets beat by UNIX & C in the '80s

UNIX platforms started to populate information technology. How did this happen? What happened to IBM's mainframe operating systems, and DEC's VMS? And what happened to the software revolution of the '70s. To answer these questions, consider the following influential events in this history.

- The U.S. Government, particularly the Department of Defense (DoD), assumed that its problems were faced by the rest of the world. These problems were caused by (1) programming in assembly language, and (2) not using standard operating systems. Most government developers were hardware-oriented. They lacked real software experience. This was evident with “embedded systems” where the applications and operating systems were wrapped together in a single development. Large complex commercial software successes were built using reliable operating systems and high-level languages. These were virtually ignored by the DoD hardware vendors who preferred to “roll their own” on time & material contracts. The end result was the launch of the Ada language effort with high expectations.
- As a policy, the Government resisted platforms with *proprietary* (privately developed) operating systems, e.g., IBM's MVS, and VM. Many of the reasons have to do with the Government's desire to own special licenses that private businesses perceive hard to control. Somehow, UNIX was considered an OPEN SYSTEM and became the operating system of choice. Large quantities of VAX computers with UNIX were delivered to the Government and its contractors, but productivity plummeted. The problem was blamed on the operating system. (What was not recognized was the use of C and C++ as the programming languages for this environment.) DEC saved the day with VMS, a new (proprietary) operating system, along with good FORTRAN and COBOL compilers. It worked well and started to become the de facto *real* standard for that period

- Computer-Aided Software Engineering (CASE) tools became fashionable in both government and commercial organizations, being marketed by a number of software houses. These tools included graphics and automatic requirements documentation. But the software problem remained unsolved. This is because CASE is left by the wayside once the coding starts and changes start to mount.
- On the government side, programmers building embedded systems for big contractors gravitated from assembler into C and C++. When the software problem persisted, DoD tried to dictate the use of its own new language, Ada. Special contracts were offered to document case histories of how Ada saved time and money. But despite its elegance, Ada did not solve the problems. Ada programmers were among the most expensive in the world. The economic realities of getting systems delivered were hard to combat, and DoD's major system program managers got waivers to use software languages other than Ada. This battle was won by the C++ / OOP crowd.
- UNIX continued to be promoted by AT&T and academia followed. The Government continued to characterize *proprietary* software (developed and licensed commercially) as not in its best interests. After VMS and other proprietary operating systems rose to great heights, they hit a water-fall drop in sales - as if they were black-listed. UNIX and C had proprietary stamps, but somehow were OK.
- Hardware vendors are motivated to sell platforms without having to invest in an operating system, and UNIX allowed users to run multiple tasks in virtual memory mode.
- Whereas INTEL chips all came with an assembler, designers of powerful workstation chips found a way to hide their instruction set architecture - behind the C-based language compilers, as C became the new intermediary language.

The above events were driven by a desire to move toward more powerful platforms at a lower price. The difficulty, as always, is dealing with the software interface. Learning and using C-based languages and UNIX required a large investment. Organizations complained about the unfriendliness of the UNIX-C environment and the corresponding rise in software costs. These complaints did not stop this movement. The low cost of powerful hardware platforms housing UNIX became attractive.

As Windows platforms started selling in large quantities, UNIX developers, e.g., SCO, SUN, SGI, and HP were able to achieve a reasonable level of compatibility with Windows application software. Under SCO's Open Desktop, one could have a Windows session going and flip back to a UNIX session. C, which came along with UNIX, slowly became as ubiquitous as COBOL. Because C was developed in the very early 70's, many of the revolutionary ideas from the following years virtually disappeared, being replaced by a new wave of thinking - Object Oriented Programming (OOP).

THE RISE OF C, C++ AND OOP (circa 1985 - 2000)

C is quirky, flawed, and an enormous success.

- Dennis Ritchie, one of the original C authors - from van der Linden, [98].

Most people in the software field believe that the underpinnings of current software development environments, namely the C and C++ languages, were developed under a well planned R&D program at Bell Labs. As pointed out in various references, see Anselmo, [2], this is a total misperception. AT&T did spend billions of dollars competing in the computer field and promoting UNIX, and C and C++ were the languages of UNIX. But the real driving forces behind C were to build a small compiler to fit in a very small amount of memory, and a spartan syntax that made the compiler easy to build, [2], [56], and [78].

C++ is a modified version of C to provide an Object-Oriented language. Object-Oriented Programming (OOP) claims to support reusability, but that definition of reuse is a technicality. OOP reuse actually inhibits real productivity by making real reuse difficult, see [1] and [83]. The OOP approach has also proven to be hard to scale as new requirements evolve. As a result, large systems built in C++ go out of control quickly unless the management approach is very intensive, with large numbers of programmers responsible for relatively small amounts of code, see [44]. This approach leads to much lower productivity. However, the amount of investment in this approach - both in careers and dollars - presents huge inertia.

Object-Oriented Programming (OOP) is defined by a set of properties that are supposedly independent of a particular programming language, see [95]. These properties constitute a set of requirements that a language is supposed to meet to be considered Object-Oriented. Today, OOP is most closely associated with the C++ language.

When reading the C-based language (C, C++, and Java) literature, one finds sets of postulates and ultra-simple examples of OOP in C++. If one looks for scientific ties to programmer *productivity* and *economic* measures, one is left disappointed. Although the word *good* is used as some kind of measure of a language, one has to ask: "What *measures* make OOP good, and particularly, good for what?" More interesting reading is that of the different OOP camps, and the objections they raise with each other's approach. Different camps appear to have their own ideas of what's *good*.

There are allusions that C++ OOP properties provide improvements in computer programs. But there is little experimental evidence, or suggestions of experiments, to support a scientific comparison. Our experience on actual projects correlates to the productivity data in Chapter 1. Based on large projects, our assessment is that it costs more, not less, to build equivalent quality software using C++ OOP. In the support phase, it is hard for one programmer to understand what another did.[†]

[†] A common joke is that C++ is a *write-only* language. One programmer writes it and no one else can read it.

Finally, many experienced people on the sidelines have said that this movement has been a great step backwards for the U.S. software industry. The feature article of a 1994 issue of *Upside Magazine*, [97], interviewed five leading technologists to get their view on the world of technology in the year 2000. The questions covered broad areas of communications and automation. One of the questions was “What advancement will be the biggest disappointment?” Surprisingly, Gordon Bell, architect of DEC’s VAX family, and John Warnock, CEO of Adobe Systems had the same answer - Object-Oriented Programming! Warnock said “I think the whole object thing is a red herring.”

Having made an investment in becoming proficient in a subject, one does not want to think of one’s investment of time as being wasted. It is hard to consider scrapping a skill that has taken years to learn, one that was supposed to provide significant economic benefits. After all of that effort, one does not want to hear that there is a better direction, especially if the alternative might involve another learning process. This creates a significant inertial factor among proficient C++ programmers.

Another Trail From There to Here

How did it happen that the C-based language technology has moved so far into the forefront? We’ll start to answer this question by referring again to Peter van der Linden’s book, *Deep C Secrets*, [98]. On page 297 he describes the desirable features of an OO language.

“... Abstraction is useful in software because it allows the programmer to:

- hide irrelevant detail, and concentrate on essentials.
- present a "black box" interface to the outside world. The interface specifies the valid operations on the object, but does not indicate how the object will implement them internally.
- break a complicated system down into independent components. This in turn localizes knowledge, and prevents undisciplined interaction between components.
- reuse and share code.”

These bullets appear to be very attractive, ones that should relate to economic benefits. We agree with them in principle, but let’s look at each of them more carefully.

- Who decides what’s irrelevant detail (and therefore hidden)? An engineer should be able to uncover the (hidden) algorithms, decide what is wanted, and if it’s correct. Material not wanted should be covered up again. Else features are inherited that are undesirable.
- A black box approach is OK after it is built, and tested. But if one has ever tried to test (or reuse) a black box that is the least bit complicated, one wants to open it up to understand it, to reduce the space of possible outcomes (tests) by many dimensions. Else, one is prone to finding out why things are not working after they are in production. To quote a "quote of the father of C++”, from *Deep C Secrets*, [98]:

C makes it easy to shoot yourself in the foot. C++ makes it harder, but when you do, it blows away your whole leg.
- Bjarne Stroustrup

- Who determines the architecture of "independent" objects? How is it measured? How does one see the *whole* picture? Independence can be accomplished without hiding.
- Reuse is desirable, provided one can see inside the box and determine what is needed. Sharing black boxes is like groping in the dark. Everything should be open to inspection to see if it warrants sharing. Else we become *overloaded* - with unnecessary baggage.

There are subtle points of confusion here. Specifically, one must be able to differentiate between hiding irrelevant details, protection from change, and reuse. *These are three different objectives that need not conflict.* As we shall see, reuse depends directly upon independence and understandability, neither of which inhibits hiding irrelevant detail or protection from change.

Apart from the OO concept, there are many reasons why C-based languages are difficult to understand. We refer again to van der Linden's book, [98], on the use of C and C++ in a production environment at SUN, where he questions the placement of the burden of translation. *Should it be on the programmer, or on the language translator?* On page 64, he states:

"C's declaration syntax is trivial for a compiler (or a compiler-writer) to process, but hard for the average programmer. Language designers are only human, and mistakes will be made. For example, the Ada language reference manual gives an ambiguous grammar for Ada in an appendix at the back. Ambiguity is a very undesirable property of a programming language grammar, as it significantly complicates the job of a compiler writer. But the syntax of C declarations is a truly horrible mess that permeates the *use* of the entire language. It's no exaggeration to say that C is significantly and needlessly complicated because of the awkward manner of combining types."

Neither sound economics nor good science can be cited as the framework for this movement. When engineers think of comparisons, they think of benchmarks that produce clear-cut economic measures. Example: *Machine X runs my job three times faster than machine Y, and this saves 20 hours a week.* But the software field appears to be void of sound economic measures or real science (repeatable tests and benchmarks). So what were the underlying driving forces at work to make a C-based language the programmers' choice?

OOP is appealing, intellectually. The fact that OOP productivity claims are not supported by experiment does not seem to matter. Many software approaches have been fostered by organizations without hard economic forces driving their research. This results in club-house or hobby-shop technology that cannot stand up to the test of a real competitive environment. The commercial market in the U.S. was still dominated by COBOL and FORTRAN in 1995. Referencing the October 1995 article in Inform, [51], studies by IDC, Gartner Group, and Dataquest show that COBOL alone still accounted for 80% of all business applications. We do not have more recent data, but the landscape has been changing rapidly since 1985. More importantly, the number of articles questioning software productivity is now growing rapidly.

If the picture were simple, it would have already been sorted out. The shroud of complexity around C-based OO languages has left management subordinate to the claims of the programmers. But cracks in the dam are appearing. Large projects are becoming less common. Software problems have cost top executives their jobs, or even forced a sale of the company. In the next few sections we attempt to shed more light on why the software problem has become worse - not better.

THE GROWING TECHNOLOGY GAP

As we described in Chapter 3, during the 1960's the financial industry had to deal with experienced programmers who insisted that accounting applications could only be written *efficiently* in assembly language. The real concern were the high school students who were cutting the cost to build software using COBOL. Data processing managers had to work hard to dislodge their assets from the hands of the assembly language programmers. It was clearly a job security fight.

The modern version of this problem was highlighted by Paul Strassmann, former Assistant Secretary of Defense for C3I, at a 1992 Ada Symposium at George Mason University. There he described the necessary transition of the software industry in his speech "From a Craft to an Industry." He presented the results of a study on the resistance to change in the mode of production of software by what he termed the "loner programmers," the people that every computer installation has come to depend on. He said

"You can easily identify them. ... They are immersed in their craft, but find it difficult to explain or document it. They usually work late into the night, trying to fix a problem caused by low quality and frequently repaired incomprehensible software. ... They place little reliance on assistance from others and most likely disregard orderly documentation and business practices... The computer code they write is unique, elegant, and usually incomprehensible to others - which explains why they are highly valued as indispensable staff."

As stated above, end users do not really care how software is built. They want systems that help them perform their tasks. Only software people make a living building software. Users make their living selecting and using systems that improve their own productivity. They don't care whether their application is totally in hardware or software. If software is used, they don't care about the language in which it is written. Just like any buyers with freedom to buy what they choose, users want high quality systems that are easy to use at the lowest possible price.

The problem faced by system developers today is one of building user friendly systems that are difficult to implement due to the complexity of options and wide range of functionality required. Most of these systems are implemented in software. However, programmers are not trained to design systems, and are not equipped to flush out user needs. They are trained to write computer programs. And so software productivity is still, today, the most significant stumbling block in building complex automated systems.

As Paul Strassmann said, most automated systems are still built as if programming were an art or craft. Programmers carve out their pieces of code and hope they can be *inherited* by the next generation. Programmers do their own designs, build the software, test it, and even answer customer questions. In most shops, there is little, if any, cross-checking or management intervention. In many shops, good programmers are called *authors*. Their desire to see a higher level of value placed on their craft presents a dilemma to the average manager of a staff of programmers when it comes time for salary reviews.

WHY IS THE TECHNOLOGY GAP WIDENING - INSTEAD OF SHRINKING?

One does not have to look hard to find articles describing dramatic reductions in the cost of computer power. Quoting an article from UPSIDE magazine, [71], "The end-user price per MIP of computer power has gone from \$250,000 in 1980 to \$25,000 in 1985 to \$2,500 in 1990, and to \$50 in 1995. Today it is more like 50¢. But as computer and semiconductor manufacturing productivity continues to move ahead by leaps and bounds, and the software productivity index continues to go lower, the hardware-software technology gap gets wider and wider.

In an article in 1989, Business Week [16] reported that "Software is the major stumbling block. For years programmers have been unable to crank out new packages fast enough for mainframe customers to get the most from their machines. Now, with multiple mainframes, dozens of minicomputers, and hundreds of PCs, customers are 'over-mipped.' They have tremendous computing capacity as measured in MIPS - the power to process one million instructions per second." "But we don't have the software to fill the MIPS" said Robert C. Hughes, vice-president for industry marketing at Digital Equipment Corp in that same article.

The article proceeds with "The key is to get programs that are MIPS suckers." "But there's a catch: It has been possible to automate the design of increasingly powerful hardware. Creating software, however, remains a laborious and slow undertaking. And the biggest MIPS suckers are the hardest to produce."

The Case Against CASE

In another Business Week article two years later, [17], methods for improving software productivity were discussed, including computer-aided software engineering (CASE) and OOP. "For years, the industry bet on CASE tools to automate software development. But a recent survey by CASE Research Corp. shows that fewer than 35% of CASE customers say such tools have improved programmer productivity or quality. 'There are a lot of people who haven't made it work yet,' concedes Mike Waters, general manager of Texas Instruments Inc.'s CASE division." In that same article, OOP was also described as a possible contender for improving software productivity. However, the article ended the discussion by stating that "most software experts warn against relying too heavily on such 'silver bullet' technologies."

Resistance To Innovation

So where is software technology headed? It appears to be moving backward -- toward the days of cryptic languages. This is apparently the result of a lack of measurement in programming languages. People tout Java and C++ as the languages of the future. In fact, C is almost as old as COBOL, being born as B from the Basic Combined Programming Language (BCPL) in the late 1960's. So age is not the difference.

The history of C, see [2] and [3], indicates that the people who invented it wanted a compiler that was easy to write and could fit in a very small computer. (B was designed to fit into the PDP-7's 8K word Memory.) It was not designed for ease of understanding. C++ was born out of a similar notation but enhanced with the theory of Object-Oriented Programming

In the first sentence of the preface of their book, **The C PROGRAMMING LANGUAGE**, [56], Kernighan and Ritchie state that "C is a general-purpose programming language which features economy of expression, ... C is not a 'very high level' language, nor a 'big' one, ..." In the second paragraph of the *zeroth* chapter (CHAPTER 0: INTRODUCTION), it states that "C is a relatively 'low level' language."

Probably more important are the abstract definitions of OOP that take old words that everyone thought they understood, and use them in a different way. It takes time to fit together all of the new definitions. For example, what is a *protected abstract virtual base pure virtual private destructor* (from van der Linden, [98])? But now you know something that clearly sets you apart. What it has to do with software productivity, in terms of saving time and money for your employer, becomes immaterial if it guarantees your job for the next few years.

The security factor is deeply rooted in basic traits of human nature. One of the strongest forces affecting the acceptance of new technology is the perception of one's job security. This creates a very strong inertial factor that resists change.

It is this gap between perceived job security and real job security, resulting from higher productivity, that has put the OOP-C technology where it is. But if management starts making accurate economic comparisons in a fair market environment, the reversal will begin. And it has started. Since 1994, the market for mainframes has come back to life. Marketeers credit this to the "UNIX After Market" - after buyers realize that the cost to port their software to UNIX and C-based languages is greater than the hardware savings gained from UNIX platforms. And it's not just the cost of the port, or of building new applications. It is heavily influenced by the cost of supporting the software over the long term.

Separation of Skills - The Requirement of an Industrial Approach

We submit that separation of skills is the biggest differentiator between a craft and an industry. The industrial revolution not only automated many jobs, it took crafts and turned them into industries. This was most apparent in factories, where different job skills were clearly classified. One did not have to be a craftsman to participate in the production of goods. One could look to a career path that moved up the line as one increased architectural or management skills. But such an environment does not exist in software. And this is what is keeping software from moving from a craft to an industry.

There is a lack of production-oriented technology in software to support separation of the skills. In other words, an environment must exist that supports the separation of skill sets. With everyone using the same set of tools there are no differences. In every other engineering discipline, there is a clear separation of designer from technician, architect from builder, etc. So why not in software? Because there have been no tools to provide this separation.

The Gap Between *Perceived Job Security* And *Real Job Security*

In a Software Special Report, [17], Business Week posed the question "Can the U.S. Stay Ahead in Software?" This article described the growing number of software engineers and programmers in other countries whose price per hour is less than 1/3 of their equivalent in this country. Thus, the cost of software development in foreign companies could be much less than their U.S. counterparts, even if they were not nearly as efficient at it.

In that same article, Lim Joo-Hong, deputy director of research at Singapore's NCB brought out one of the major factors - "Software only needs people. There is little need for a lot of other resources." In that same article, Edward Yourdon, publisher of the monthly newsletter *American Programmer* in New York warned that cheap labor abroad could begin to make low-level programming jobs in the U.S. obsolete. He was quoted as saying "The only thing that has prevented it from becoming a crisis so far is that the software industry is growing so fast that we haven't seen many jobs taken away." The article further warned "Without such entry-level jobs, the U.S. won't be able to employ large numbers of computer science graduates, further discouraging careers in the field."

Yet, in the U.S., when productivity is an issue, we divide into camps. One camp says "What should we do to shed fat and cut costs?" Another camp says "What should we do to produce more with our current resources?" And a third camp starts to form a union to insure that salaries will not be lowered, younger workers cannot be hired at competitive prices, and anyone with seniority or longevity can't be laid off.

The approach to be taken certainly depends upon the market situation. If the market is stagnant or drying up, we better slim down and get more productive at the same time. If the market is good and possibly growing, we must become more productive, but investments to get there can help the bottom line by increasing market share. In neither case does the union mentality hold hope for the long run. Historically, it constrains management to make decisions that are not economically sensible. This causes a gap between perceived job security and real job security. Real job security only improves when an organization becomes more competitive, i.e., when that organization becomes more productive relative to its competition.

THE NEED FOR - AND INERTIA AGAINST - A NEW TECHNOLOGY

The Structure of Scientific Revolutions by Thomas Kuhn, [57] is a classic work that describes the nature of scientific revolutions and corresponding paradigm shifts. History is replete with major breakthroughs that have been stymied by politics for decades before they were accepted. Change of any reasonable amount must be justified by a sufficient "quantum leap" of improvement before it can be accepted. Another work by Clayton Christensen, [29], *The Innovator's Dilemma*, tells about the fall of great companies when significant innovation comes about, typically by small companies. It is first resisted by the large companies - and their clients, both having large vested interests in maintaining the status quo. This is termed a "Disruptive Technology." It must be worth the upheaval from an entrenched approach and the corresponding investment in retraining.

Change can be promoted swiftly by the glaring failure of existing approaches. In the software field, failures are becoming more apparent. The disparity in productivity in the computer field between hardware and software is an excellent example that is forcing the need for change. Like the ripples before the major wave, change is preceded by trials that provide feedback into an approach that fills the real need. And that is what we perceive to be the situation in the software field today. The early waves have occurred because of the obvious growing need for change. But the real wave has yet to hit the shore. The old mold is not yet broken.

The Case for a New Paradigm

For about 30 years, the Moore's curve held that the number of transistors on a chip would double every 18 months, and therefore, so would computer speeds. Then in 1997, Dr. James Meindl of Georgia Tech predicted it was going to flatten. Anyone buying computers can validate his prediction. In fact, Intel's Itanium chip, with its 64 bit address space, is capable of handling a huge leap in memory, but its internal clock speeds are actually slower. As memory appears to be headed beyond our dreams, processor speeds are stalling.

But this problem can be overcome. Computers are being built today with thousands of processors, and using a single operating system to maintain speed, see [26]. They contain hundreds of terabytes of memory. Also, the ability to handle the cache coherency problem¹ in these computers appears to be under good *hardware* control. So things still look rosy for expansion in the computer field.

But look again. While hardware engineers produce great feats, tearing down barrier after barrier, looming problems in software have been hiding behind them. Quoting Marcus Ranum, [77],

“...I see that Microsoft, Intel, and AMD have jointly announced a new partnership to help prevent buffer overflows using hardware controls. In other words, the software quality problem has gotten so bad that the hardware guys are trying to solve it, too. Never mind that lots of processor and memory-management units are capable of marking pages as nonexecutable; it just seems backward to me that we're trying to solve what is fundamentally a software problem using hardware. It's not even a generic software problem; it's a runtime environment issue that's specific to a particular programming language.”

But Ranum's article is just one example. There is a large group of people experienced in both sides of the computer field - hardware and software - saying the same thing. More importantly, the statistics on productivity show that the software industry has been going downhill every year. But that is not the worst of it. To take advantage of a large number of parallel processors requires a new approach to operating systems. And here, it seems that we can't even get it right for a single processor.

¹ Ensuring that data stored in one processor's local memory is not “out of sync” with that of another processor.

Recent articles about Microsoft's problems with its new Vista operating system bear this out. In the Sept. 2005 Wall Street Journal, Guth, [46], describes Microsoft's delays in its effort to come out with Vista, a new version of the Windows operating system. It quotes Microsoft executives saying that there was no architecture!

This article was followed up by Cusumano in the ACM, [32], where he talks about the gridlock occurring on the Vista project, stating:

“We now know that the chaotic ‘spaghetti’ architecture of Windows ... was one of the major reasons for this gridlock. Making even small changes in one part of the product led to unpredictable and destabilizing consequences in other parts since most of the components were tied together in complex and unpredictable ways.”

It now appears that we are already into an era where the computer field is constrained by software problems that present barriers to using new hardware technology. To counter this problem, we must do a reversal on our approach to developing and supporting software. And this approach must be based upon a new paradigm that takes full advantage of all that hardware.

What does it take to break the mold? It requires a thorough cleaning of the conceptual slate of computer programming paradigms that we follow today. It requires a hard look at the history of evolution, of how we've moved from writing in ones and zeros to where we are now, analyzing the initial waves of a revolution. It requires a careful distillation of the changes that have occurred, the improvements in productivity, and most important, the underlying causes for these improvements.

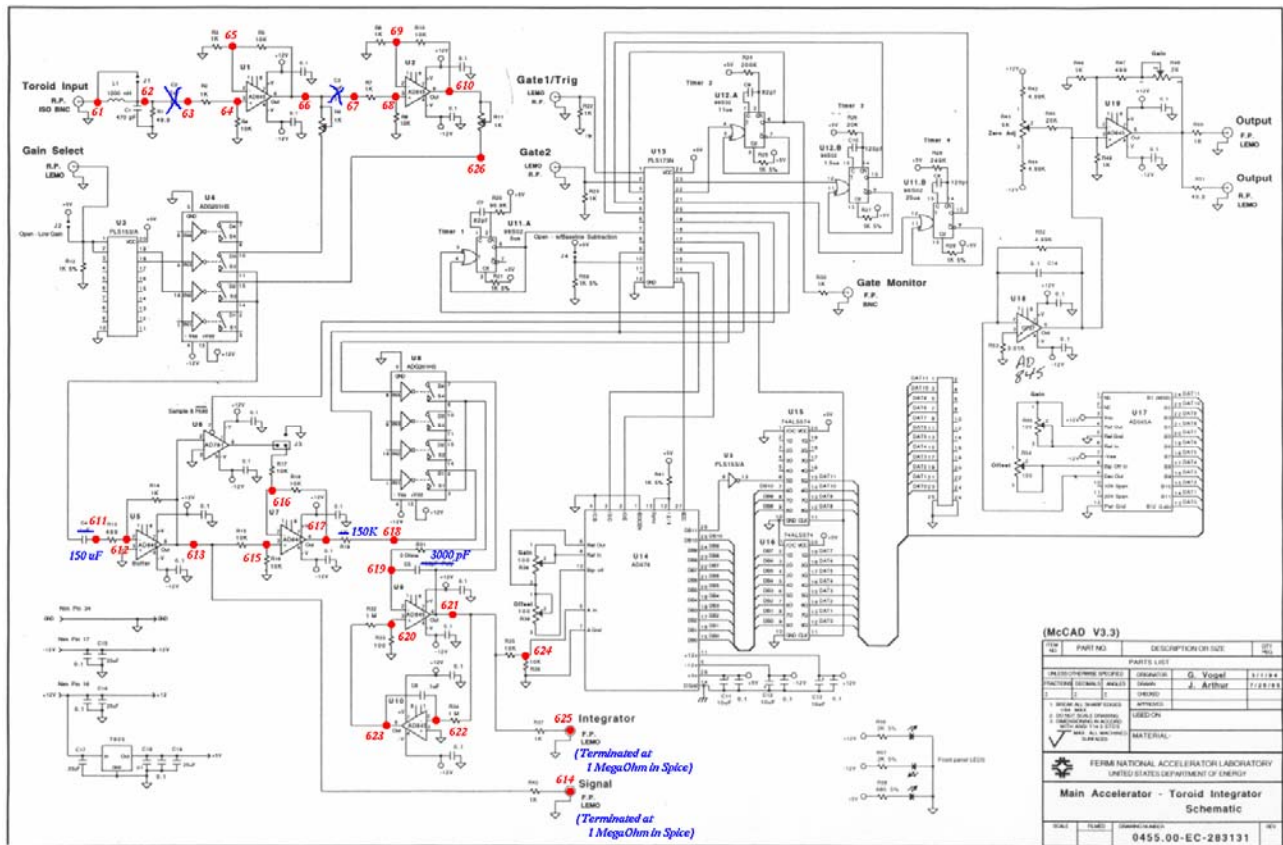
In commercial software, one need not write programs or code for many applications anymore. Systems have been developed that build user interface panels, and generate code automatically from input charts. Typical financial reports can be generated using similar automated systems. One need not be a programmer to use these systems.

In highly technical fields, e.g., communications or computer engineering, system engineers typically work to understand user requirements and produce automated system designs based upon powerful hardware facilities. These engineers would just as soon have the programming part of the project eliminated. One solution: automate the software development and support process. But how can we do that? As indicated above, we already have in many areas. Many of these areas involve Computer-Aided Design (CAD) of very special engineering functions. In this book, we are seeking a similar path for software. It separates design from implementation by separating architecture from language. Separate skills are then required to perform these different types of efforts, and the language is easy for a subject area expert to understand. And this brings us face-to-face with the *real* problem.

As described in *Microcosm* by George Gilder, [40], human inertia is the major deterrent to innovation. Some seasoned programmers have characterized the new technology presented here as “unprofessional.” When asked why they consider it unprofessional, they typically reply, *anyone can do it!*

In the chapters that follow, we describe a new paradigm for building software systems. It allows developers to build models of a system in a simulated environment using graphics and high level languages. Using this approach they can convert their models directly into actual system modules and then watch the real thing. It follows the paradigm used for CAD of hardware systems. It presents a new way to build large complex software systems, allowing one to build live test drivers in a simulated environment to exercise the actual software modules.

New students of software immediately relate to the ease of use of this new approach. They particularly like the ability to quickly build sophisticated graphical representations of information - something they are otherwise not able to do in today's classrooms.



Chapter 5. Objectives Of A Software Environment

WHAT ARE WE TRYING TO ACHIEVE?

We know of no successful software application that has not evolved with changes once put into the hands of its users. If new software works well, then as soon as the users start using it, new requirements emerge. If the developer does not accommodate the desired changes, someone else will be waiting in the wings. To survive in this competitive environment, software product developers must have their next upgrade in the hopper, in parallel with the one just going out, even though it may be months to another release. Allusions to *security* and *long lasting* do not equate to standing still in a world of economic freedom. In today's competitive environment, the quest for *survival* implies *constant improvement*.

Lowering the time and cost to develop a product does not necessarily imply that it will cost more to support. It is the overall life cycle economics that one must be concerned with. Getting an initial product into the hands of users quickly is a well tested strategy by many successful software companies. Being able to get a product out fast may also imply that we have the right tools, and the people that know how to use them. This can actually cut the cost of both development and support. In fact, software history is filled with the reverse case: many very costly developments have ended up with software that could not be supported.

It is our thesis that if a software environment does not favor ease of design and rapid prototyping, it will not support products that survive in the real market. Implying that these features are incompatible with long-term, high quality software development appears to us as unjustified as the small memory model. Software is built by people - architects and implementers. If they are provided with the proper environment, including knowledge of customer economics, and knowledge of the tools and assets at their disposal, they will make the best economic decisions.

We start by considering software properties that correlate to minimizing the time and cost to build and support complex systems. The definition of these properties has evolved from many years of empirical evidence that we have accumulated building complex systems and simulations. The most desirable properties of the modules comprising a software system are:

- Reusability
- Scalability
- Understandability
- Independence

We seek an environment that ensures these properties are realized in the resulting software in a way that produces the desired economic benefits. This implies an engineering approach to the design of a high productivity software development and support environment.

The concept of reusability has been important to the justification of various OOP paradigms. This is a very important concept to us, in that reusability generally saves time while improving quality. However, our definition of reusability is surprisingly different from that used for OOP. We must understand this difference so we can differentiate our paradigms from those of OOP.

THE MANY DEFINITIONS OF REUSABILITY

In the OOP paradigm there should be no need to have programmer-A, who wants to reuse programmer-B's class (module), look within the module to discover how programmer-B has implemented the functions in that class. For the purposes of our discussion, a class can be thought of as a software module.

Inheritance As An Approach To Reuse

If programmer-A wants to use programmer-B's class, he incorporates it as a subset of the class he is building. If he wants to change the meaning of a function in the "reused" class, then he can define the implementation of the named function within the current class and it will automatically take on the new meaning. However, he will not have access to the functional implementation of the original class. This inhibits changes by anyone other than the original author - a protection mechanism.

We quote Kenneth Rubin, [83], in *Encapsulating Change*:

"By limiting the knowledge of how a particular function is performed to one place, we promote reusability and shield the system from the effects of change. When change does occur, its effects can often be limited to the inside of a single message. As long as the object still behaves outwardly the way it always did, the rest of the system will be unaffected. Brad Cox put it well in his book, *Object Oriented Programming, An Evolutionary Approach*: 'Objects build firewalls around change.' "

Using this approach, software is built having one black box inherit another, and so on. With just a few layers of inheritance, the unused baggage that must be carried can become excessive. To alleviate this problem, one may use special linking loader facilities to insure that only those functions actually used in the program will be linked in.

The paper by Rosen, [81], provides excellent examples of the problems with inheritance, challenging basic OOP concepts. He cites the lengthy interchange among programmers on OO bulletin boards regarding the reusability of classes. Clearly, inheritance has to be convenient, i.e., it has to save time and money over the life cycle of a piece of software if it is to be a desirable property. Rosen also points out the relative nature of measures. Almost any high level language is much better than writing in ones and zeros. Therefore, anyone could say that all high level languages are *good*, or at least *much better*.

Thus we must qualify the words *good* or *much better* relative to a reference frame to ensure they are meaningful. On this basis, Rosen draws the conclusion that "The popularity that OOLs (*object-oriented languages*) have achieved necessarily means their use carries a number of benefits. Many of these benefits, however, are most noticeable only when compared with older programming languages such as Pascal or C."

Rosen's paper shows how different people within the OOP community look at reuse differently. He dissects the problem of inheritance as a means for reuse that allows a programmer to bend the inherited properties within his own class to meet his design criteria. We fully agree with his conclusions on how this leads to a string of tightly connected dependencies that are difficult to maintain.

A More Careful Look At Reuse

We encourage the reader to stand back and observe how the waters are easily made murky by arguments that are not tied to real economic measures, based upon real software experiments and their outcomes. This has led to different definitions of reuse that are on different direction vectors relative to the economics of software life cycles. Specifically, measures of software economics are not invariant to the hardware environment - the computing machine itself. The availability of huge amounts of memory at very low costs has changed the way programmers justify their time. We predict that parallel processing power will provide even greater dislocations of economic reasoning, once a software environment becomes available that is as friendly as that of a single processor.

Let's take a practical example of reuse of a complex software module, e.g., one that manages large databases for direct access. One typically devises a method for rapidly retrieving keys in main memory that can then be used to retrieve records directly from disk. One prefers not to start from scratch to write a linked list, or other lookup method to manage the key index. Typically, one cannot reuse the same module in the sense described in Rosen's paper.

Most often, 5% to 25% of the design must be changed. Given a good design, it typically takes much less time to change and test an existing module than it took to build and test that part of the original module in the first place. Using the modular approach that we prescribe, one simply copies the old module, and starts making the changes. Even if both modules happen to be used in the same task, concerns about duplicate code are typically insignificant when compared to other factors that swamp out the economic effects.

For example, duplicate code merely means more memory is being used. So what? One must compare the price of memory to the time it takes to change, test, and support a design so it can use old code. Many cases that we are familiar with, involving a reasonably large module, would be sidetracked into becoming a relatively large project, just to figure out how to meet the requirement without changing the existing module. In fact, the solution could end up convoluted and hard to understand by anyone other than the original author. This would lead to significant cost increases in the support years.

Moreover, it is typically only the instructions that end up in the duplication issue. The data buffers are usually independent. But, the instruction set is usually small compared to the database. One merely has to compare instruction swap space to data page space utilization in most programs to see this.

The Large Memory Model

If one is concerned about speed, then duplication wins hands down. Trading memory for speed is what today's large memory model machines are all about. In retrospect, the modular approach permits one to go down inside a module, pull out the relevant submodules, and create a new module, an architectural option. It allows the architect to consider more aspects of the design, create more easily reused submodules, and maintain a good architecture, one that will support future changes. The architect is the best one to make this decision, on a case-by-case basis. This normally results in an all-around enhancement when completed.

We must emphasize that the hardware world has moved to the *large memory model* today - leaving the small memory model behind. There was a time when memory designers talked of the limitations they faced in terms of size, speed, and cost. But today we have it all! It's big, fast, and cheap, particularly when compared to software development costs. Writing software to fit into small areas of memory is a poor use of resources. Trying to manage memory better than today's virtual memory managers is also a poor trade. In fact, unless you turn off the system memory manager, it will take your manager, slice it up, swap it and page it back and forth between the hierarchical hardware memory layers, as it does its own thing.

Inheritance Inhibits *Ease of Reuse*

Proponents of OOP lead us to believe that “black box” reuse via inheritance is a discipline to which programmers should adhere. Based upon our experience, this type of discipline does not correlate to improved software quality or productivity. The problem is this:

- *inheritance inhibits both understandability and independence, the two major factors affecting real reusability.*

Rosen's concern with inheritance supports our view. Anything *connected* to an old module represents a dependence that is potentially messy. Dependencies must be well understood, else the new module will be hard to extract and test as well as to maintain.

As an example, assume that we wish to improve a module. With inheritance, and the consequent hidden code, we may have little knowledge of the implication of a change. To gain that knowledge, we may need to see everything connected to that module. This does not inhibit protection mechanisms that ensure unwanted change will not occur.

We have no objections to hiding code that we do not wish to see, since it may cloud the issues of concern. But if we wish to understand what is going on inside a module, we may need to see it all.

Inheritance may inhibit independence. If an old module is inherited, the new module may have dependencies on the inherited module that are invisible to the new module. If we inherit the new module somewhere else, the problem is compounded, as these *hidden dependencies* cascade.

WHAT ARE THE REAL OBJECTIVES?

We believe one of the major objectives is a open environment, where visibility and protection are both very high. We fully understand the importance of tightly controlling changes in a production environment. This involves making decisions based upon business goals as to what should be changed, deciding on priorities, and managing access to those modules that have been authorized for change. This must be followed by highly disciplined regression testing, including controlled changes to, and growth of, the test driver packages.

To implement the open approach, software modules must be *protected* from unauthorized access. But once the access is authorized, those given the responsibility for change must be able to drill down to any level of detail required to understand the logic. This includes having access to the complete details of relevant modules that they are not authorized to change. Experience shows that visibility leads to better solutions, including a potential management decision to change something not previously authorized. It is common to decide that the real culprit is a different module that, if changed, makes the problem easier to solve, and solves similar future problems.

As another case, one may decide to *copy (and rename)* a module not authorized for change. Then that module can be changed to solve the original - as well as other - problems for different modules. This does not affect the independence or protection from unwanted change of the original module. *Controlling change is a protection issue - not a visibility issue!*

Another objective is the ability to isolate problems quickly. One should not have to plow through layers of modules or code to find out they are irrelevant. This implies the ability to quickly perceive where the problems are and what must be added or changed. This is an architectural issue. Experienced hardware designers have learned to inject layers of isolation between modules to ensure their independence.

The property of independence is directly applicable to software. This becomes obvious when one can “see” the architecture. But one must first realize that architecture can not be seen from the code. This implies that the architecture must be visualized or *drawn*, an engineering approach. This also implies that we must have an unambiguous - *one-to-one* - mapping from the architecture to the implementation (code), just as we do with drawings in other engineering disciplines.

The Importance Of Understandability And Independence

In a production environment, one would not reuse a module without verifying its operation. This can be done in two ways. One can take a black box approach (OOP) and validate the input-output relationships. This implies running all tests necessary to learn what will happen with all possible inputs. But, as illustrated above, inheritance causes the level of complexity to go up exponentially with the ancestors. Thus the dimensions of the test space can become enormous quickly as the concatenated sets of black box responses grows, making testing very difficult.

Alternatively, one can limit verification tests by having a detailed understanding of the internal design of a module. This leads to the first most important property from our economic standpoint - *understandability* - of the details of a module.

In the world of computer simulation, where validity is a hard constraint, one could not sell a model on a "black box" basis. Engineers must be able to easily understand what is inside the model so they can validate it from a specification standpoint. They want to review engineering drawings, and read data structures and rule structures, as if they were written specifications. They want the ability to get in and make changes easily, to suit their own needs.

Our objective is to provide the same open view in software modules. This in no way implies lack of protection of the original module. Nor does copying a module imply the creation of an unnecessary redundancy within a given system. Most of the original module may be used as a utility, maybe by many other modules where appropriate. We want our development teams to be concerned about the trade-offs of getting the initial product built - using minimum time and resources - versus the efficiency of the end product in terms of running times and memory utilization. Most important, we want our teams to know that if good initial products are built, they will be reused over and over - provided they can be easily understood, validated, modified, and supported by other than the original teams that built them.

The fact that a module may depend on many other modules in a tightly coupled manner, leads to the next most important property - *independence*. When a module is independent, it has minimal coupling to other modules. This implies it shares only those attributes required for its specific function. At this point, we must point out the practical problems versus the theoretical problems, and take an engineering approach. If everything is working properly, independence can be ignored. It does not affect the operation. But make a change, or encounter a problem from a user, and independence leaps to the forefront.

As indicated above, independence is an architectural design property that cannot be appreciated unless one can see the architecture. An architecture that provides a high degree of independence of modules provides for fault isolation, an important property of hardware design. It also supports understandability, since the scope of concern is limited.

A REVIEW OF SUCCESSFUL ENGINEERING PRACTICES

When skyscrapers, bridges, and aircraft are built, the probability of failure must be reduced to extremely small percentages. Architects and designers of these systems must produce reliable design plans and accurate time estimates to avoid what become highly visible mistakes or failures. This is generally accomplished by reusing previously proven designs. To lay the ground work for a new approach to software, we will first review some long-standing engineering practices that have evolved to protect against project failures. See, for example, Jesse Poore's article, [73].

Engineering Drawings

In electrical, mechanical, aeronautical or architectural engineering environments, the requirement for *design reusability* is so obvious that it goes unmentioned. In these environments, a design, and its corresponding specification documents, are controlled by engineering drawings. When people buy a design, the most critical piece they buy is the set of engineering drawings. The drawings graphically depict the architecture. The specifications provide the details.

Prior to its automation, engineering drawing was a required course in most engineering curriculums. It is a discipline for carefully representing complex architectures pictorially. It is implemented as a formal method of documentation, complete with standards for controlling multiple sets of drawings, interconnections between drawings, revisions, etc. Drawings provide for references to more detailed drawings in a controlled hierarchy, as well as sets of carefully written specifications. The intent of the drawings and specifications is to allow someone other than the original designer to build to the drawings and specifications.

The most critical difference between engineering drawings and graphical approaches to software is that engineering drawings are a precise depiction of the connectivity of the physical entities being described, *with no ambiguities*, relative to the desired specification level. If one reviews engineering drawings of circuits, chips, boards, machines, etc., every icon and interconnect line represents a physical element.

If one reviews graphical depictions of software, e.g., block diagrams, state diagrams, bubble charts, structure charts, Booch diagrams, etc. everyone contains *layers of abstraction with ambiguities* within the desired specification level, i.e. there is no direct mapping to the elements of code.

This suggests that we need an engineering drawing approach to software that eliminates these ambiguities and maps the elements of a drawing right to the code. The approach that achieves this objective is derived from the *Separation Principle*, developed in 1982 by Cave, [23]. The Separation Principle separates data from instructions, a basic paradigm switch from programming languages, and particularly from OOP. It provides the framework for drawing software architectures, automating the design process, and changing the paradigm of programming as we know it. It provides for visual design reviews of the engineering drawings that get right to the heart of problems before they get out of control. Engineering design rules can be checked by a quick visual scan, or even automatically. We know of no other approach that does this. It is described in detail in the chapters that follow.

Separation Of Architecture From Detailed Implementation

In hard engineering disciplines, architecture is separated from construction or fabrication. Engineers design the architecture and then oversee the construction or fabrication process. They don't bend the metal, solder the joints, or drive the nails. Engineers may not possess these skills, skills that often require craftsmen that have been trained over a number of years. Masons, carpenters, plumbers, electricians, sheet rockers, painters, etc. do not meet and decide to build a building. They are given a detailed set of architectural drawings and specifications to follow. The architect oversees the construction of his design to make sure that his drawings and specifications are being properly interpreted and followed.

There is no such separation of skills in the programming world today, particularly with OOP. Object Oriented Programmers get together and decide on the objects to be built, and who should build them. Then they proceed to build them. (A common joke in the industry is that interfaces are designed by the first programmer that gets there.) It has been said by many people who lived through the software revolution of the '70s that OOP killed top-down design in one shot. As a result, it has killed the concept of system architecture as being a separate discipline from writing code in a programming language, a problem emphasized by Strassmann, [94].

The approach that we advocate provides the natural separation of skills, between architecture and detailed implementation, as in other engineering environments. Architects produce the drawings and top level specifications, people skilled in constructing detailed data structures and algorithms complete the final product in a somewhat natural language. No one writes computer code using a typical programming language, e.g., C++ or Java. C and Open-GL code is generated automatically. By raising the level of understandability and visibility, the importance of architecture and its effects on reusability become apparent.

Designing Reusable Modules

To design reusable modules, we must decompose the functional parts of a system and organize them into a convenient framework that provides visibility of architecture and clarity of detail so that people other than the original designer can easily understand it. In hardware, this is done by creating a design that breaks the system down into a well structured set of independent modules, i.e., they can be tested and supported independently from the rest of the system.

The most important properties of good *reusable* software modules are the following:

UNDERSTANDABILITY

- Visibility - of the separate functional aspects of each module, the interdependencies of each, and thus the structure of the system;
- Clarity - of detail, as well as structure, so that they are quickly understood, revised, and enhanced;

INDEPENDENCE

- Isolation - between the parts to minimize their interdependencies;
- Protection - of modules from being inadvertently changed in a way that would cause other modules not to operate correctly (regress);

These properties also determine the *scalability* of a system, i.e., its ability to expand to provide much wider functionality.

To achieve these properties, one is faced with the problem of achieving visibility and clarity while providing isolation and protection. From the OOP perspective, these desires may appear to conflict with each other. The resulting OOP solution appears to ignore visibility and clarity. Such a solution fails when one builds very large systems.

The key facilities required to support the modularity properties are:

- Separation of architecture from detailed implementation.
- Graphical depiction of the architecture
- Languages that support the above two bullets

These are introduced below.

COMPUTER-AIDED DESIGN AT THE *SYSTEM* LEVEL

Automation of programming has already been accomplished successfully in the field of Computer-Aided Design (CAD). In the 1960's, engineers who required complex simulations to support difficult electronic designs were highly dependent upon programmers to implement the models and simulations. Today, engineers build very large complex models and run simulations using CAD systems without anyone writing computer programs.

Using the approach described here, engineers describe the software architecture and algorithms for complex systems in their own terms, using graphics and natural language. This is a CAD approach to building software systems, a vision described by Jesse Poore, [73]. An illustration of this approach is shown in Figure 5-1. It is taken from a product called VisiSoft that we describe later.

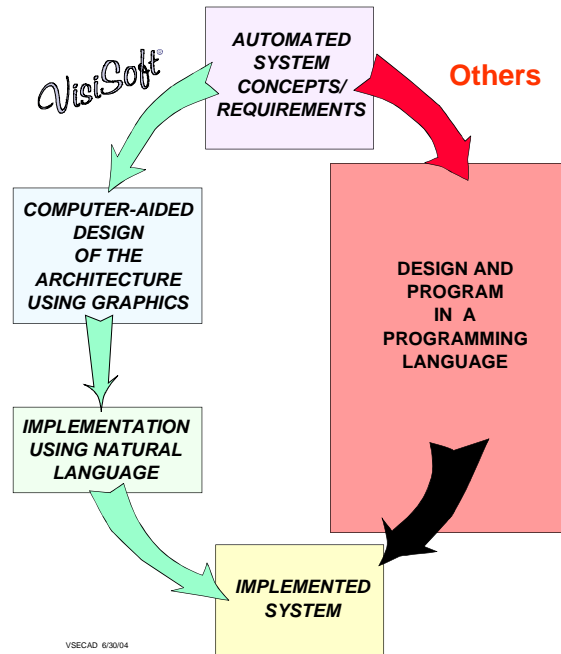


Figure 5-1. Computer-Aided Design (CAD) at the system level.

For drawings to provide a precise definition of the desired engineering design solution, they must be free of abstractions or ambiguities at the level of specification required. An example is the graphical depiction of the logical or electrical design of a system. One sees various types of gates and flip-flops interconnected with specific wire connections. There are no abstractions, and therefore no ambiguities, between the design and implementation level. A flip-flop may be implemented in many ways at the electrical level, and one can argue that the symbol of a flip-flop represents an abstraction from the detailed electrical design standpoint. However, at either the electrical level or logical design level, there are no ambiguities.

In OOP, the design is implemented at the programming language level, with the classes or objects being defined in the language. There is no architectural view of the structure, no detailed drawings, only descriptions of functions or messages that are associated with classes of intentionally hidden code in a language, e.g., C++.

The same freedom from ambiguity should exist at the architectural design level in software. It is our assessment that current diagrammatic approaches to depicting software, including those of the CASE and UML systems, have a layer of abstraction between the top level design and the detailed implementation (code). This is because of the ambiguity of what the interconnections depict.

To provide a CAD approach, the interconnection of well specified blocks of code must be defined precisely by the drawing, without ambiguities between the architecture and the code. Our intent is to show the reader that we now have the same level of CAD facilities available to support designers of general software systems as those used for engineering design in other disciplines. This is CAD at the *system* level, not just the chip or board level as it currently exists.

Architecture Versus Detailed Implementation

Engineering CAD systems serve a particular design or implementation purpose using an integrated set of tools. This is apparent in computer system design where one has different CAD tools to support logical design, electrical design, chip mask design and printed circuit board design. These design tools have been integrated to eliminate design ambiguities as one moves from the logical layer to the electrical layer to the chip and the board. They are used by experts at the different levels, although the experts at the electrical level generally understand the design problems at all of the levels.

Software organizations tend to lack this hierarchy of expertise. There are applications specific experts, and “systems” experts (OS, communications, etc.). But there is no hierarchy from the specifications to the implementation of an application. The approach offered here introduces a precise separation of the *design* of a system's *architectural structure* from the *detailed implementation* of its *elementary modules*. This provides a focal point between architecture and implementation. There is no doubt about who is responsible for what. The structural architecture requires an expert who knows how to design large complex systems. The detailed implementer has expertise in implementing modules - more of a coding function.

This separation provides for management control of the total design through an architecture environment, and the corresponding engineering drawings that are produced. Design reviews are at the drawing level. It does not take knowledge of a programming language to understand whether a design is good or bad. An architect can generally look at the drawings and determine if the design structure is good, without getting into details of the implementation of a specific module. We note that, as in engineering, architects generally know both. But electricians and plumbers, although certified in their skills, are not architects.

Support For The Software Life-Cycle

Figure 5-2 depicts the typical steps in the life-cycle of a software product. A successful product gets used for many years. It evolves to meet new user needs during its lifetime. The figure shows the iterative steps to building and supporting a system. If the design takes this life-cycle of constant change into account, then the architecture must evolve to support the changes. As in hardware, this implies breaking the overall system into individual modules within a system that are maximally independent. The shaded area shows the focus on this type of design. One must continuously reconsider the overall system architecture as well as the individual module architectures that implement the evolving design.

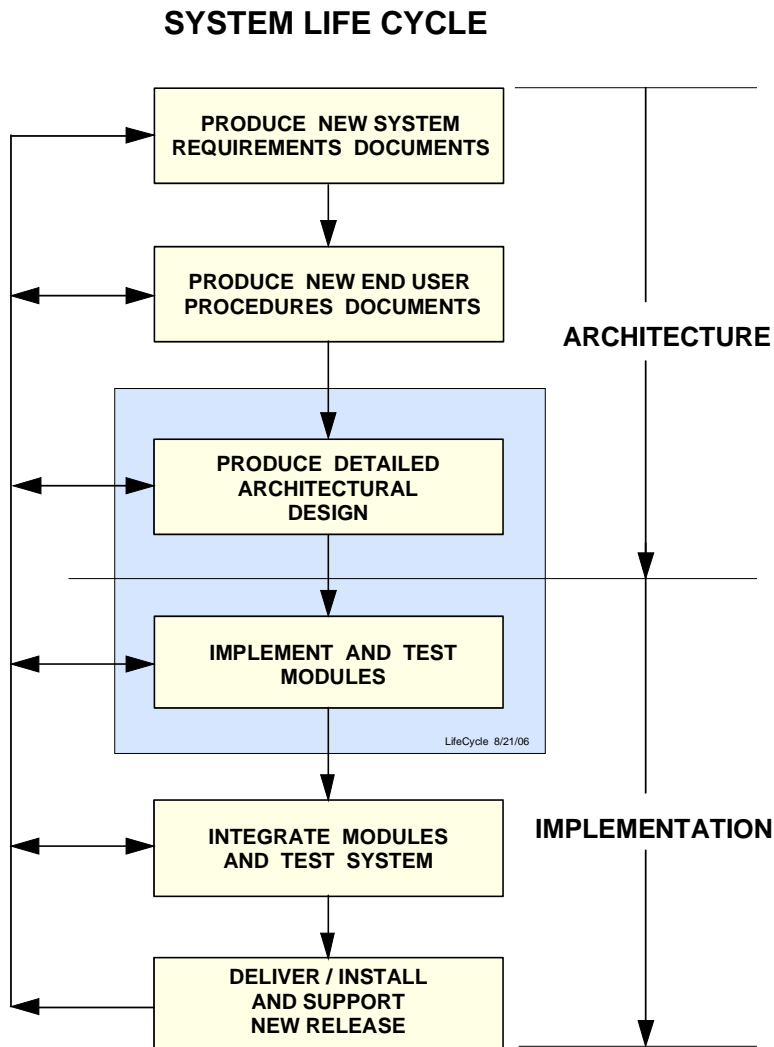


Figure 5-2. Typical steps in the incremental development of an evolving system.

To insure reusability of modules and the ease with which a system can be understood, one must produce an architecture that supports ease of implementation of the attribute and rule structures of the modules. This is no different from constructing a set of complex buildings in a real estate project. One would not consider calling in the carpenters, electricians, plumbers, etc. without a set of architectural drawings that provides these people with all the details needed to ensure that everything they implement will go together to form a high quality integrated product.

It is essential that the overall architecture of a system be well designed. For example, large systems may be split into multiple tasks, each being highly independent pieces that simplify development and support. Separate tasks may run on separate processors, communicating via multiple networks. Deciding how to split the system into separate tasks is clearly an architectural design question. Breaking out the top-level modules of a task into submodules is also a major architectural issue. This requires that critical interfaces are specified at the module architecture level.

One must specify, if not design, some of the detailed layers to the level of the data structures at their interfaces, and specify what rule conditions will be needed, before finalizing the architectural design. This implies an iterative approach to the design as shown by the arrows in Figure 5-2. Having an architecture that accommodates change is necessary to accurately estimate and control a new release.

As detailed implementation proceeds, and more information is understood about real system details, one may determine that architectural improvements are needed as certain modules evolve. As a system starts to produce output results, and one finally faces what is important, it may be necessary to significantly enhance, or even add, modules in the architecture.

A system must be designed so that these problems are accounted for at the beginning, not when someone is expecting final results. We believe that the most significant factor affecting the cost of structural modification is the manner in which the architecture is designed. It must be designed so that the inevitable forces of change are easily dealt with, accommodated, and controlled. This is covered in Chapter 8, "Software Architecture".

Our approach is tailored to support the ease of understanding and evaluation of system, task, and module architectures through the use of engineering drawings. This insures the development of architectures that permit major changes in task and module structures with relative ease. To take advantage of this facility, one must apply the appropriate discipline to the architecture phase of a system. The detailed architecture must be completed first, and a set of drawings reviewed and approved. The drawings must be inspected to determine if the design provides for module independence and ease of restructuring.

Those who have developed multiple successful software products know that the constant requirement for change is a major factor in the life cycle of complex software systems. For them, the lesson is obvious. The software survivors will be those with the ability to control increasing complexity, and this requires the ability to adapt to continual change.

We agree with the basic premise of DeMarco and Lister in *Peopleware*, [36]. The cultural and sociological environment in which creative people must work is important. Their attitudes must be directed positively toward solving problems that require intensive thought and concentration. Providing these people with the proper tools and management environment is critical to maintaining that attitude. And, as Deming implies, happiness (survival) stems from constant improvement - on all fronts.

Designers and implementers must understand that their value depends upon economics as defined by the buyer of the software they produce. This must be perceived from a life cycle standpoint. Everyone must be well informed about customer desires, satisfactions, and complaints. The attitude of the designers and implementers toward survival is important. If they can relate the survival of the software they are building to their own personal value and survival, then they are in the best position to make the economic decisions on how to do both the architecture and the implementation.

IMPLEMENTING REUSABILITY

By our measures, reusability of a module is determined by the following factors.

- Range of Functionality
- Scalability
- Understandability
- Independence

The wider the *range of functionality*, the more likely that a module can be reused in a new application. This is obvious from questions like "Does your module contain ... ? If not, what will it take to incorporate it for my application?"

Scalability simply implies that the range of functionality is easily expanded. One must consider this from the standpoint of adding new functions as well as expanding an existing function.

A high degree of *understandability* simply implies that no special knowledge is required to understand the module description. Anyone with a good knowledge of the particular application being developed should be able to understand these descriptions. In other words, one should not have to learn a new language to determine whether a module is valid and desired. One should only need to know what the module is supposed to do functionally.

Independence is determined by the ease with which one can pull out an old module and replace it by a new one, or test it independently in a separate test jig. This, in turn, is determined by the amount of connectivity that the module has with other modules in the system, and can be determined by the number of interconnections between modules as shown on the engineering drawings. This is hardly different from the measure of independence of a hardware module.

Productivity

From the above, we conclude that productivity depends upon understandability and independence. We have drawn an illustration of these two factors, postulating their relation to productivity in Figure 5-3 below.

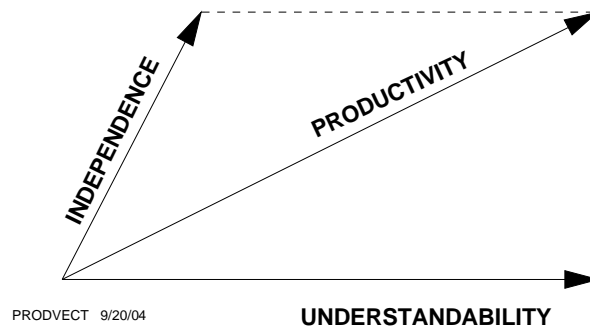


Figure 5-3. Relationships among productivity, independence, and understandability.

Clearly, there are a number of additional factors that correlate to productivity, and also to independence and understandability. The purpose of the illustration is to show simply that productivity relates to the components of independence and understandability. We admit that assigning quantitative values will depend on many things, and may be difficult to measure. But we must begin the measurement process.

Looking at the above figure, one can perceive understandability as being measured in man-hours for someone, other than the original author, to understand and change a given module in the support mode. One can perceive independence as being measured in man-hours to independently test the changes to that module. Productivity is measured in man-hours to complete a change to a module, i.e., to modify and test it. The time to change the module will depend upon its independence as well as its understandability, and this is where orthogonality must be considered. We advocate experiments that would further quantify these properties, and believe such research efforts would result in very important contributions to a *real* science of software engineering.

As we will see in later chapters, using the VisiSoft environment, we can get a quantitative measure of independence in terms of a connectivity matrix, i.e., a measure of what processes (instructions) share what resources (data). We can also take economic measures of the time it takes to independently test modules with different levels of connectivity, in isolated test jigs. Understandability could be measured along the lines of Fitzsimmons and Love, [38], and Ledgard et al, [59]. All we are trying to accomplish here is to articulate the hypothesis that productivity increases directly with increased independence and understandability, and to suggest approaches to the scientific experiments required to establish its validity.

THE NEED FOR AN ARCHITECTURE ENVIRONMENT

OOP does not have a true architecture environment. Everything is done inside the programming language. Although Rational Rose and Borland offer an environment, they are built around C++ where architectural design functions are performed inside the language. This assertion will be made clear in the next few chapters.

Achieving Scalability Via Hierarchies

As indicated in Chapter 2, *scalability* is a measure of software size that can be achieved under full control. It also determines the ease with which a software product can continue to grow - under control. Scalability is a major driving force in promoting good architecture. The reasons stem from basic principles of controlling complex systems and organizations, particularly those that are growing and changing.

The military provides an excellent example of the problem of an organization maintaining control of its assets under extreme change - even near chaos. It is well known that such systems are maintained and changed using hierarchies of control. In the military, this is known as the "chain of command." It is highly organized for quick decisions and rapid responses. It is not a bureaucracy (they can be flat organizations). Hierarchies are also key to controlling the design and manufacture of complex engineering systems. This is most obvious in the aerospace industry for the construction of large jet liners.

Using hierarchies to achieve scalability is also obvious in software, once the paradigms are in place to observe them. Unfortunately, in the current software field, hierarchies appear to be an anathema. In fact, they are a critical characteristic of complex data structures, complex rule structures, and complex architectures that are scalable. This will be apparent in later chapters.

Graphical Depiction Of The Architecture

Our approach draws on experience with hardware modularity and the use of engineering drawings to provide visibility of the design hierarchy. This includes a measure of module independence that ties directly to the time and cost to test, support, and reuse a module. This requires that the architecture be precisely defined by the drawings, with no ambiguities. It requires a clear focal point as to what is architecture and what is implementation.

To accomplish this requires that we have graphical visibility another layer down, to the interconnection of data and instructions. This can only be accomplished if we can review a one-to-one mapping of each, implying *data and instructions must be distinct, separate elements on a visual diagram*.

Separation Of Data From Instructions

Figure 5-4 contains the layout for one of IBM's PowerPC RISC chips. It is representative of today's modern machine designs. There are separate Instruction Cache and Data Cache, Instruction Tags and Data Tags, Instruction Memory Management and Data Memory Management, etc. The same instruction sets can act on different data sets. So, when a compiler generates assembly language, and the assembler generates machine code, data is separated from instructions for management inside the machine on a very general and dynamically natural basis.

Understanding the internal processes of a computer helps us to take advantage of the technology. For example, data gets operated upon by instructions. Instructions are fetched from memory and typically cause the values of the data portions of memory to change. We may add one memory area to another memory area, putting the result back into one or another memory area. It may get copied from one area, possibly transformed using registers and the arithmetic/logical operation units, and the result put into the same or another memory area. *Data does not flow from one spot to another. It gets transformed as a function of time.* If X is a generalized "data vector," then one can write $X(T+1) = A*X(T)$ where A is a generalized transformation on the data vector of interest. If one considers external inputs as $U(T)$, then one has $X(T+1) = A*X(T) + B*U(T)$, a well-known general dynamic operational form.

Taking this concept one step further, consider that a data vector is a subset of the total data set (memory area) available to a task. If X is the data vector available to a particular *process* (set of instructions) within the task, then $X(T)$ is the state of that data vector before it executes, and $X(T+1)$ is the state of that data vector after it executes. The operator A represents the transformation on that data vector as process A executes between time T and $T+1$.

Reprinted with permission from /AIXtra, IBM's Magazine for AIX Professionals.

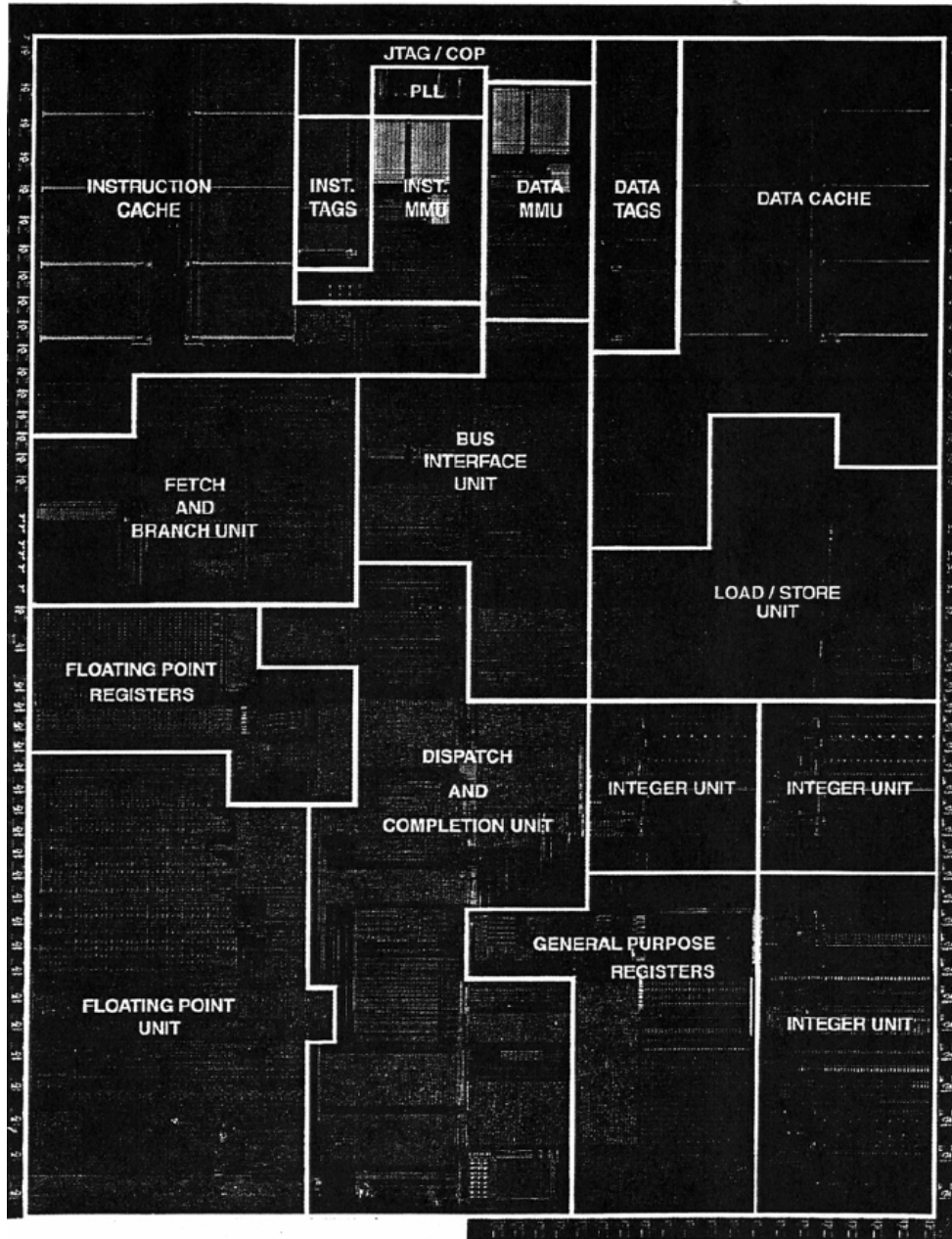


Figure 5-4. The IBM PowerPC 604 RISC chip.

We now define generalized data vectors as *Resources* that may contain strings of characters or words as well as numbers. *Processes* transform resources from state to state. We call this a *generalized state space*.

A CAD APPROACH TO SOFTWARE

Figure 5-5 illustrates a CAD concept applied to software. This figure illustrates module hierarchies, with some modules covered - others showing details. Modules are uncovered with a mouse click. At the detailed level, small ovals represent hierarchical data structures (*resources*) and the small rectangles represent hierarchical rule structures (*processes*). To have access to the resources, processes must be connected by a line. Note that some resources are shared, having connections to more than one process. Others are dedicated to a particular process. Colors indicate type of module. The module shown in the figure is a library module, called from other modules.

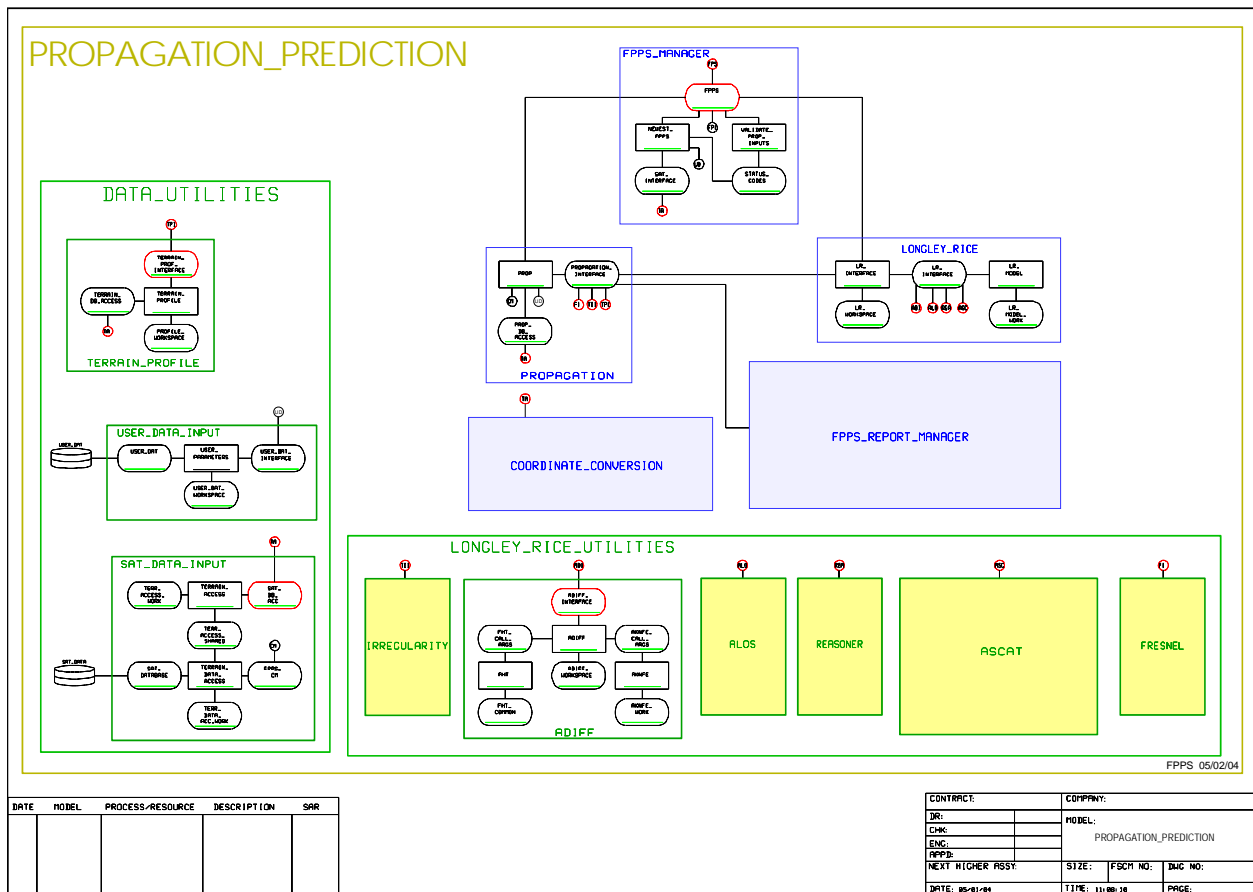


Figure 5-5. Illustration of primitive data elements connected to primitive instruction elements.

This approach requires a complete paradigm shift from current computer programming methods. It provides a one-to-one mapping from the architecture to the primitive language elements with no ambiguities regarding how they are connected. In Chapter 6, we will show that this paradigm is derived directly from that used for representing general dynamic systems in control system engineering. We shall also describe the profound implications that this approach has on the separation of architecture from language, and the ability to insure that only the architects can change the architecture; implementers (coders) cannot.

This approach was driven by the need to develop large scale simulations of communication and control systems, simulations that must run very fast - on parallel processors under a single operating system. This led to a breakthrough - now called the “separation principle” - that separates data from instructions. Conceived in 1982 by Cave, [55], this approach allows one to track software module independence, and automatically allocate processors to processes at run-time on a large parallel processor. Module independence is determined by the number of external connections between a particular module and other modules in the system. This can be determined by visual inspection of the drawing.

The separation principle provides the basis for engineering drawings of software, with a one-to-one mapping from the drawings to the code, a true form of *software architecture*. As in other fields, architecture is much more graphical than algebraic or textual. Whether designing machines, ships, or buildings, architects produce drawings. These drawings are not “approximate” or suggestive, but precise engineering specifications used directly in production. Until now, drawings of software were abstractions to aid in design, but not of much use in the actual production or support mode. Just as important are the separate languages that define the data and instructions. They are read easily by non-programmers, a requirement for subject area experts validating complex models.

This CAD environment has evolved over hundreds of successful software and simulation projects, and is now a fully integrated product for developing software. Prior to this CAD approach, software architecture did not exist, an observation that is immediately apparent upon seeing it. After using it, one quickly draws the analogy between current programming approaches, and architects in other fields trying to produce designs without drawings. It is readily apparent that one cannot determine the independence of software modules without the ability to see a picture of the architecture. The recent articles about Microsoft’s problems with its new Vista operating system bear this out, [32], [46].

Architecture Versus Flow Charts

In Chapter 3 we noted the introduction of flow charts during the era when programmers wrote in machine code or assembly language. This was because the statements on the flow chart were more readable. In particular, instructions that transferred control took time to understand. Furthermore, a single flow chart element typically represented multiple instructions. A single box may have had an equation in it, and a test and transfer diamond could split in three directions. However, as FORTRAN and COBOL appeared, flow charts became cumbersome compared to the code because the statements were more easily read and understood. A good language covered as much on a line as did an element on the chart, and took a lot less paper.

The disadvantages of flow charting became more pronounced with languages having more readable names and good control structures, eliminating the use of the GOTO. This did not stop people from wanting flow charts, and programs appeared that created flow charts from COBOL code. The classic joke was that flow charts were created automatically and attached to documentation - but no one ever looked at them. Productivity was clearly improved using a good language instead of diagrams of what the language was saying. Flow charts diagram the language inside a process, not the architecture as described here.

Generalized Data Vectors And Transformations

Our use of the term drawing in conjunction with architecture has nothing to do with flow charting (or similarly state diagrams). Flow charting represents the flow of control in a routine or program. Most software people who have worked on large complex data systems for a long period of time will quickly point out that it is the data that drives complexity. The manner in which one breaks up the database, comprised of all of the elements used to make decisions as well as used in operations, is key to the architecture. Flow of control follows.

As described above, using state space equations in engineering (or the equivalent), systems are represented as transformations that take place at a point in time. If the transformations are properly designed, then the changes unfold in time as desired. The systems are not represented in terms of a flow diagram. They are represented in terms of a state vector and transformations on that state vector. The architectural approach defined here is basically the same, except that we have extended the concept of a state vector to a generalized state vector (a resource) where symbols and words are not restricted to numbers, and transformations (processes) are not restricted to mathematical operations. These are implemented using data structures and rule structures respectively.

The Drawing - Language Breakpoint

There is clearly a transition point between language and drawings. Where should that point lie? We offer the following generalizations. Drawings will be used where they improve productivity over language. Language will be used when it improves productivity over drawings. We note that productivity as used in this book is defined in Chapter 2, being measured using loaded costs and life cycle time measures in a competitive product environment.

In the building industry example, architects provide sets of written specifications with their drawings. The drawings are the top level control system. They reference the specifications for more details. So just as productivity suffers using all language, it would also suffer using all drawings.

We have described the productivity advantages of flow charts when writing in machine code or assembler, and the loss of those advantages when higher level languages were introduced. This was particularly true with the advent of COBOL, with its understandable names and dramatically improved control structures. This is indicative of a breakpoint between language and drawing.

A flow chart could apply to the language inside a process as defined above. But, as indicated, flow charts stopped being used when good languages were developed. With more understandable names, good control structures, and no GOTOs, flow of control is best represented using language statements, as opposed to diagrams.

But flow charts are a totally different concept than the drawings described here. The architectural breakup of a system's total database - and the transformations on subsets of that database - are best represented using drawings that cover a hierarchy of modules within the architecture. As we shall see, this also supports a top-down design approach.

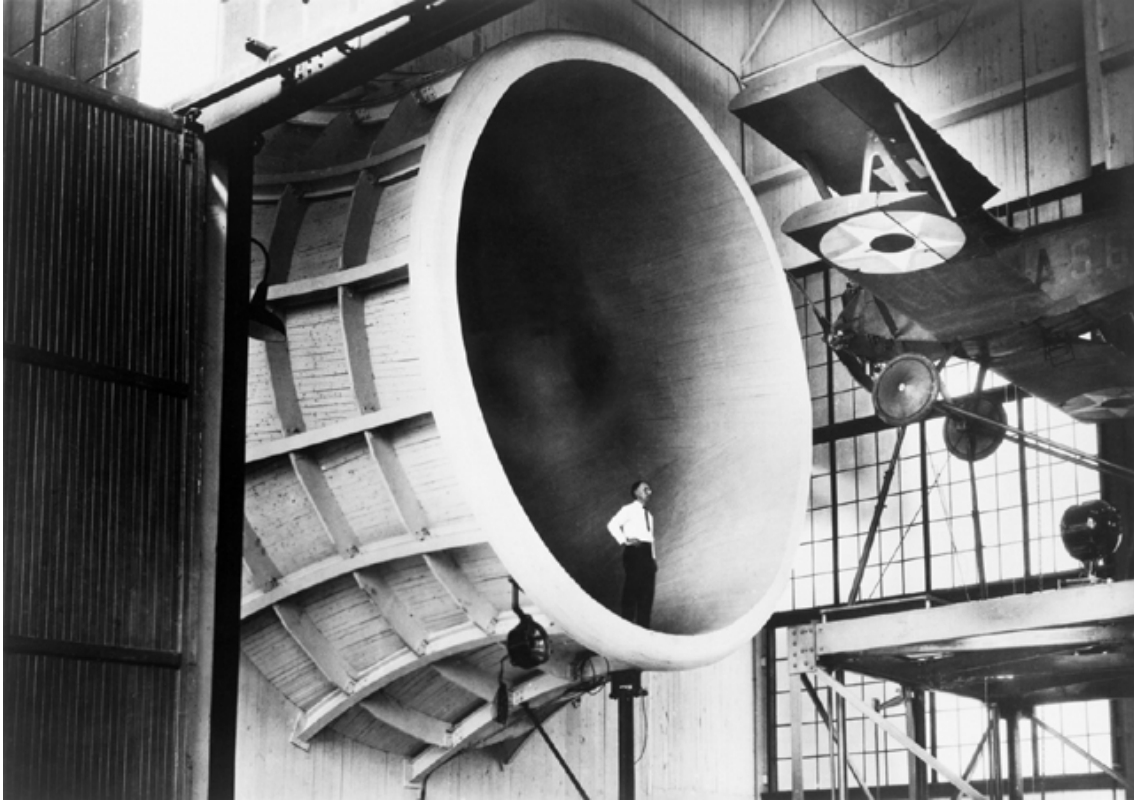
A LANGUAGE ENVIRONMENT TO SUPPORT THE NEW PARADIGM

We must now consider a language environment that supports the other new paradigm - separation of architecture from implementation. As mentioned above, this can only be accomplished by the separation of data from instructions. This is accomplished with another significant paradigm shift. Instead of a single language, there are three: one to describe data structures (data), one to describe rule structures (instructions), and one to specify controls and interfaces for tasks that represent the combination of modules in a “executable program”. This allows designers to focus on the features of each language to ensure clarity of detail, and eliminate the need to learn esoteric programming or control languages. As derived in Fitzsimmons and Love, [38], and suggested in Ledgard et al, [59], this implies that they must read like a natural language. The use of a separate general language for control, instead of special control languages (e.g., scripts, JCL, etc.), provides for independence from the hardware platform and operating system, allowing one to easily move software from platform to platform, e.g., from Windows to Linux.

By natural language we imply a language that is easily understood by anyone who can read a book and understands the subject area. One should not need to learn an esoteric language. To do this, ambiguities must be qualifiable by the context of the statement, alleviating the reader from having to parse special “mechanical language” qualification structures. Of course, using a less restrictive language shifts the burden of qualification in the translation process from the human to the computer. This implies language translators that are extremely complex. If designed and used properly, a good language provides redundancy, a significant property of the English language. Specifically, it raises the probability of quickly transferring the meaning to the reader, increasing productivity.

Part 2

A New Technology for Software Development



Chapter 6. Evolution Of A New Technology

THE EARLY DAYS

The technical backgrounds of the VisiSoft developers were the creation of Computer-Aided Design (CAD) products for electronic circuit design in the 1960s. In the early 1960s, engineers typically went to the computer center to have programmers build computer simulations to test their designs. In many cases, engineers completed the design and tested their circuits in the laboratory before the simulations were completed. This was due to the level of complexity of the modeling, software design, implementation, and test effort required to build a sufficiently accurate simulation.

The process of developing circuit simulation software was apparently esoteric for the programmers and took much too long. The results were also unreliable. As a result, engineers started to design their own CAD packages. They determined that they could automate the process, ensure more design reliability, and simplify the user interface at the same time. To them, computer programming was simply a means to an end. Their approach was to eliminate as much of it as possible so they could quickly create simulations to solve their design problems. *Their goal was a significant improvement in productivity in a world of growing complexity.*

Today, designers using CAD packages no longer interface with programmers. They interface directly with CAD systems. Engineering drawings of their circuits are built using simple graphical inputs to the CAD system. The drawing is translated automatically into simulation software and also serves as the principle documentation of the design. What once took many months of work is now done in hours.

In 1968, the creators of VisiSoft developed the first worst case circuit design / optimization software package, the Constrained Optimal Design (COD) system, [22]. Instead of analyzing a postulated design, the total design problem was defined in this CAD system. The computer posed the designs, thousands of times a second, and picked out the best solution.

In the 1970s, these creators also developed one of the first graphical interfaces to their CAD package for electronic circuit design. It allowed engineers to pull up drawings of standard circuits, modify a few components graphically, store a new circuit drawing, and call on the optimization process to design it. Almost everything was done interactively through an easy-to-use graphical interface.

In the late 1970s these same creators developed a CAD system for modeling business markets and predicting demand for products. Through the early 1980s, this package - the General Stochastic Modeling (GSM) system - was sold to large industrial organizations to support complex market modeling.

EVOLUTION OF A SOFTWARE DEVELOPMENT TECHNOLOGY

the General Simulation System

In the early 1980s, the same creators moved to solving engineering design problems for advanced wireless communication systems. When describing the complex decision processes used in communication and control systems, it was very difficult to use a mathematical framework. Modelers wanted to be able to insert IF ... THEN ... ELSE ... conditional statements anywhere in a model - easily. This led to the creation of a new CAD product for discrete event simulation in 1982. Instead of building a simulation language, they created the General Simulation System (GSS) that provides the user with a total environment.

Another requirement for this system was to allow engineers to describe their models in a hierarchy, similar to the design of other engineering systems (computers, power distribution systems, aeronautical systems, etc.). It was determined that complex decision rules and algorithms had to be expressed in a readable natural language that engineers designing these systems could easily understand. The intent was to avoid the use of esoteric programming constructs. This would ensure understandability of the models, so that an engineer without a programming background could validate the models.

More important than language was the lack of speed and scalability of discrete event simulation packages at the time. These deficiencies were the principal force in developing GSS. Users complained that existing systems could not support more than 30 to 50 complex entities in a simulation without running into executable size limits. More importantly, as entities were added, simulation running times went up exponentially. Because of these size and speed limitations, engineers went back to writing complex simulations in FORTRAN.

Even with FORTRAN, running times were on the order of 5 to 7 days to run a 2 hour scenario for a large communication system of 300 or more entities. As a result, running time became a major design constraint for GSS. It was determined that many simulations would have to be run on a parallel processor. It was decided that GSS must run very fast and be highly scalable, whether or not simulations were run on a parallel processor.

Independence - The Parallel Processor Requirement

The parallel processor requirement implies being able to allocate processors to processes (groups of instructions) that can run in parallel. Being able to determine which processes can run in parallel requires knowledge of the *independence* of the processes. Independence is determined by the data shared between processes. *If two processes share data, they cannot run concurrently, i.e., they are not independent.* Independence is an important property in most scientific disciplines, and it is equally important in software.

Separation Of Data From Instructions

In 1982 it was determined by Cave, [23], that to create a map of shared data (also called a process connectivity matrix), data and instructions must reside in separate elements. This phenomenon has since been called the *separation principle*, [55]. In the remainder of this book, we implicitly make the case that this principle provides the keystone to software engineering. As a result of this determination, GSS was designed so that:

- Data is stored in Resources that contain hierarchical data structures.
- Instructions are grouped into hierarchical rule structures called Processes.

Examples of a GSS resource and process are shown in Figures 6-1 and 6-2. Note that the resource (data structure) language contains attribute types that support the readability of the process (rule structure) language.

The separation of data from instructions came to provide enormous benefits that are not immediately apparent. First is the ability to concentrate on creating separate languages that are more natural - one for describing hierarchical data structures, and one for describing hierarchical rule structures. The construction of these correspondingly more complex translators is accommodated easily because they are implemented separately.

Hierarchical data structures are directly translatable into data records for files, or message structures for communications. This is because - What You See Is What You Get - in memory. There is no “padding for word boundaries,” another unnecessary language constraint removed by today’s computers. Memory is organized in exact accordance with the data structure created by the designer, and documented in the data description language. This allows one to move huge character strings into a highly structured set of fields that act as a template over the memory, saving much time as well as simplifying the code.

05/10/91		G S S RESOURCE REPORT		USER-ID: PSI	
RESOURCE NAME: SUBSCRIBER_ATTRIBUTES					
USING PROCESSES: PLACE_CALL					
TOTAL_SUBSCRIBERS		INTEGER	INITIAL_VALUE	0	
SUBSCRIBER_COUNT		INTEGER			
SUBSCRIBER_PARAMETERS QUANTITY(200)					
1	OFFICE	INDEX			
1	CALLERS_PLAN	STATUS	PLACE_NEW_CALL		
			RETRY_CALL		
1	SUBSCRIBER_TYPE	STATUS	DATA		
			VOICE		
1	SUBSCRIBER_STATUS	STATUS	BUSY		
			FREE		
CURRENT_CALL_PARAMETERS					
1	SOURCE	INDEX			
1	DESTINATION	INDEX			
1	SUBSCRIBER	INDEX	INITIAL_VALUE	0	
1	OFFICE_NUMBER	INDEX			
1	CALL_TIME	REAL			
1	CALL_DURATION	REAL			
1	SIGNAL_TO_SUBSCRIBER	STATUS	BUSY		
			CONNECTED		
1	PHONE_NUMBER	STATUS	UNKNOWN		
			FOUND		
1	CONNECTION_ACTION	STATUS	DISCONNECT		
			CONNECT		
CALL_ATTRIBUTES					
1	CALL_INTERGEN_TIME	REAL	INITIAL_VALUE	12	***MINUTES
1	AVERAGE_CALL_DURATION	REAL	INITIAL_VALUE	4	***MINUTES
1	VARIANCE	REAL	INITIAL_VALUE	1	***MINUTES
1	RETRY_INTERGEN_TIME	REAL	INITIAL_VALUE	4	***MINUTES

Figure 6-1. GSS Resource - SUBSCRIBER_ATTRIBUTES.

The instructions consist of natural language statements that are grouped into “rules”. The rules follow the concept of one-in-one-out control structures, precisely as defined by Harlen Mills, [66], but never available until now (with the exception of COBOL that had deficiencies in this regard). This makes it easy to build more complex rules that are readily understood by anyone who knows the application, Detailed algorithms are understood easily, independent of the original author. Both resources and processes can be fairly large while maintaining tight control over the software. This provides for huge increases in speed as well as scalability.

05/06/91	G S S PROCESS RULES	USER ID: PSI
PROCESS NAME: PLACE_CALL		TIME UNITS: MINUTES
RESOURCES: SUBSCRIBER_ATTRIBUTES		INDICES: SOURCE
	SUBSCRIBER_PBX_INTERFACE	DESTINATION


```

PLACE_CALL
  IF SUBSCRIBER_STATUS(SOURCE) IS FREE
    EXECUTE ATTEMPT_CALL
  ELSE EXECUTE RETRY_LATER.

ATTEMPT_CALL
  IF CALLERS_PLAN(SOURCE) IS PLACE_NEW_CALL
    SET PHONE_NUMBER TO UNKNOWN
    EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND.
  EXECUTE MAKE_CALL

LOOK_UP_NUMBER
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF OFFICE(DESTINATION) NOT EQUAL TO OFFICE(SOURCE)
    SET PHONE_NUMBER TO FOUND.

MAKE_CALL
  CALL CONNECT_SUBSCRIBER USING SOURCE
  SET SUBSCRIBER_STATUS(SOURCE) TO BUSY
  SET SUBSCRIBER_SIGNAL TO PLACE_CALL
  SCHEDULE RECEIVE_SUBSCRIBER_INPUT NOW
    USING SOURCE, DESTINATION

RETRY_LATER
  SCHEDULE PLACE_CALL IN EXPON(RETRY_INTERGEN_TIME)
    USING SOURCE, DESTINATION

```

Figure 6-2. GSS Process - PLACE_CALL.

After using this new approach, it became apparent that the major benefit resulting from the separation of data from instructions was the ability to provide engineering drawings of the software, with a one-to-one mapping from the drawing to the code. This is illustrated in Figure 6-3. Resources, such as SUBSCRIBER_ATTRIBUTES in Figure 6-1, are shown as ovals. Processes, such as PLACE_CALL in Figure 6-2, are shown as rectangles. Lines must be drawn to connect a process to a resource, implying that, without a connect line, the instructions in that process have no connection to the data. An *elementary* model or module, e.g., SUBSCRIBER in this case, is a blue outlined box containing interconnected resources and processes. *Hierarchical* modules, such as OFFICE contain elementary or other hierarchical modules.

Getting to a fully interactive graphical system to support all of the features and functions that can be brought to bear took years to evolve. We will skip that history here, focusing on the top level features. For example, double clicking on resource or process icons brings up an editor, allowing one to change the code as illustrated in Figure 6-4. There is no other way to build or access the code except through the drawings. Architects create the drawings. Implementers (coders) may or may not have access to change the architecture. Using GSS, large scale simulations are now developed using a fully integrated engineering drawing facility.

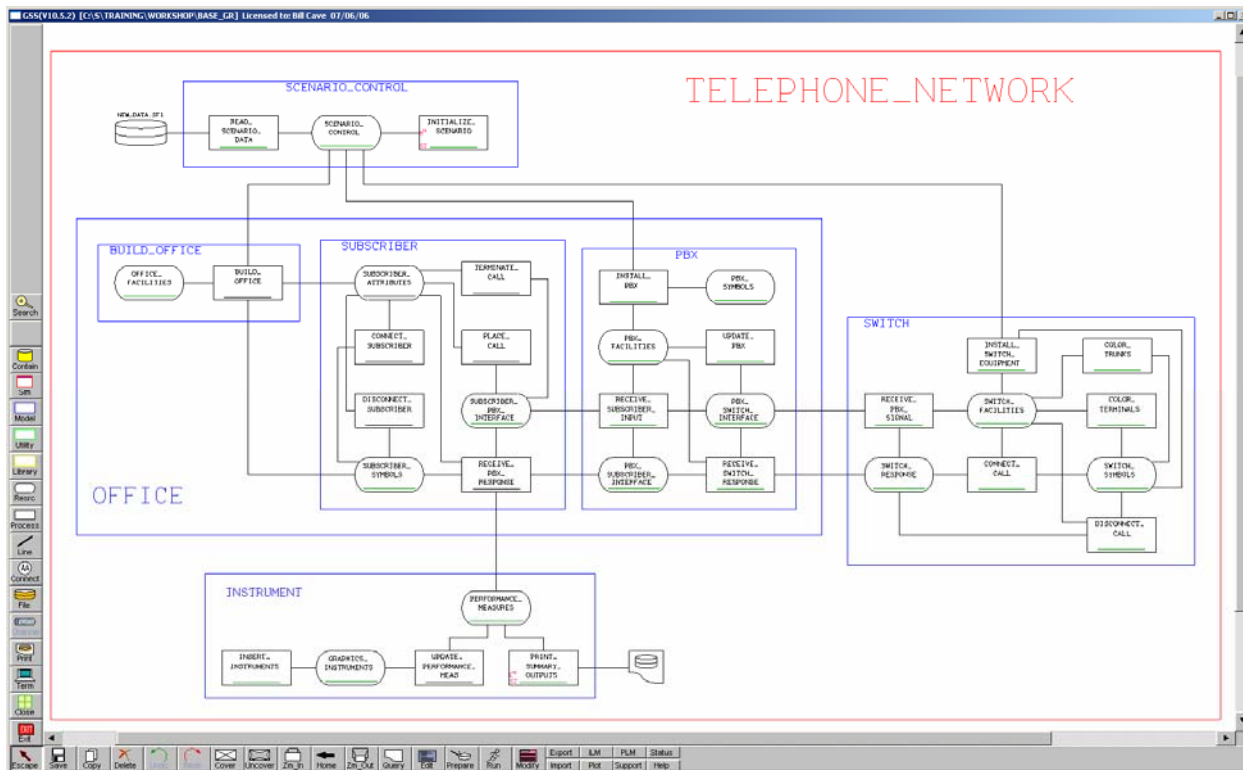


Figure 6-3. Engineering drawings of software.

```

PLACE_CALL
1 PLACE_CALL
2 IF SUBSCRIBER_STATUS(SOURCE) IS FREE
3   EXECUTE ATTEMPT_CALL
4 ELSE EXECUTE RETRY_LATER
5 ATTEMPT_CALL
6 IF CALLERS_PLAN(SOURCE) IS PLACE_NEW_CALL
7   SET PHONE_NUMBER TO UNKNOWN
8   EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND.
9   EXECUTE MAKE_CALL
10
11 LOOK_UP_NUMBER
12 DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
13 IF OFFICE(DESTINATION) NOT EQUAL TO OFFICE(SOURCE)
14   SET PHONE_NUMBER TO FOUND.
15
16 MAKE_CALL
17 CALL CONNECT_SUBSCRIBER USING SOURCE
18 SET SUBSCRIBER_STATUS(SOURCE) TO BUSY
19 SET SUBSCRIBER_SIGNAL TO PLACE_CALL
20 SCHEDULE RECEIVE_SUBSCRIBER_INPUT NOW
21   USING SOURCE, DESTINATION
22
23 RETRY_LATER
24 SCHEDULE PLACE_CALL IN EXPRN(RETRY_INTERGEN_TIME)
25   USING SOURCE, DESTINATION
26
  
```

INDEX	RESOURCE	TYPE	INITIAL_VALUE
1	TOTAL_SUBSCRIBERS	INTEGER	0
2	SUBSCRIBER_COUNT	INTEGER	
3	SUBSCRIBER_PARAMETERS	QUANTITY(1200)	
4	1 OFFICE	INDEX	PLACE_NEW_CALL
5	1 CALLERS_PLAN	STATUS	RETRY_CALL
6	1 OFFICE_NUMBER	INDEX	DATA
7	1 SUBSCRIBER_TYPE	STATUS	VOICE
8	1 SUBSCRIBER_STATUS	STATUS	BUSY
9			FREE
10			
11			
12	13 CURRENT_CALL_PARAMETERS		
13	1 SOURCE	INDEX	
14	1 DESTINATION	INDEX	INITIAL_VALUE 0
15	1 SUBSCRIBER	INDEX	
16	1 OFFICE_NUMBER	INDEX	
17	1 CALL_TIME	REAL	
18	1 CALL_DURATION	REAL	
19	1 SIGNAL_TO_SUBSCRIBER	STATUS	BUSY
20	1 PHONE_NUMBER	STATUS	UNKNOWN
21			FOUND
22	1 CONNECTION_ACTION	STATUS	DISCONNECT
23			CONNECT
24			
25			
26			
27	27 CALL_ATTRIBUTES		
28	1 CALL_INTERGEN_TIME	REAL	INITIAL_VALUE 10 ***MINUTES
29	1 AVERAGE_CALL_DURATION	REAL	INITIAL_VALUE 4 ***MINUTES
30	1 VARIANCE	REAL	INITIAL_VALUE 1 ***MINUTE
31	1 RETRY_INTERGEN_TIME	REAL	INITIAL_VALUE 4 ***MINUTES

Figure 6-4. Editing resources and processes directly from the drawing.

Independence From The Operating System As Well As The Hardware Platform

The nature of the environment in which GSS was developed was driven by clients who were buying large scale simulations for analysis and design of complex communications and control systems. In the early 1980s, this required use of DEC VAX machines running under the VMS operating system. To understand what was happening while simulations were running required an advanced graphical facility - the Run-Time Graphics (RTG) system - for visualization. RTG ran on the Silicon Graphics (SGI) workstation under IRIX - SGI's version of UNIX. It was built using the first version of Open-GL. Users could interact graphically with the simulation using RTG on the SGI machine which was networked to the VAX where the GSS simulations were running. Interactive graphics became very important to the clients, and they bought both platforms to get what they wanted.

About the mid-1980s, IBM came out with the RS-6000, which contained SGI's chips on fast graphics boards, and ran Open-GL on AIX, IBM's version of UNIX. At the same time, clients were moving away from VMS to UNIX. More importantly, GSS and RTG were selected "Best-Of-Breed" by IBM who paid to have these systems ported to the RS-6000. This port to the UNIX environment also put the total package on the SGI work-stations.

As the port to UNIX was being completed, Intel was dramatically increasing the power of its PC chips. At the same time, vendors were making fast graphics boards, and commercial versions of UNIX were becoming more adaptable to the PCs. GSS and RTG were soon running on SCO-UNIX on PCs with special graphics boards and corresponding graphics accelerator software. Then came SOLARIS on Sun workstations and finally Windows NT, 2000, and XP on PCs. The platform prices were dropping rapidly.

The Third Language

What came out of this experience was the need to be independent of both the platform and the operating system (OS). This led to a software requirement to isolate the operating system dependencies, including platform differences. This requirement was already being satisfied by the CAD approach being used to support multiple simulation runs, optimization, and graphics. These CAD features were supported by a third language that eliminated the need to write control scripts or JCL. It translated user-friendly database descriptions, graphical requirements, and multiple run requirements - including initialization - into special routines that were compiled and linked at run time as well as the control scripts for a particular platform and OS.

This third language - the Control Specification language - is illustrated in Figure 6-5. It provided many features to support simulation and optimization, database handling, graphics, and interactive control. It eliminated the need for "JCL" or "scripts", providing a development environment that is independent of the OS or hardware platform. Each task or simulation has its own control specification that is the same for every platform or operating system. Once GSS and RTG are up on a platform, one can move large complex simulations with graphics to that platform without change (the only thing that changes is run-time speed). This is a substantial factor in productivity improvement.

```

05/06/91          G S S CONTROL SPECIFICATION      USER ID: PSI
CONTROL SPECIFICATION NAME: TELEPHONE_NETWORK

CONTROL SECTION
  TITLE, SIMULATE TELEPHONE SYSTEM GRAPHICALLY
  SIMULATE

LIBRARY SECTION
  BACKGROUND

GRAPHICS SECTION
  ACTIVATE
  WORLD_SPACE LOWER_LEFT = (0, 0), UPPER_RIGHT = (1280, 1024)
  BACKGROUND_COLOR = DARK_BLUE
  INITIAL_WINDOW LOWER_LEFT = (-100, -100), WIDTH = 1280
  ICON OFFICE_OUTLINE = OFFICE,          SCALE(1.0, 1.0)
  ICON MAN             = MAN,            SCALE(1.0, 1.0)
  ICON PHONE          = PHONE,          SCALE(1.0, 1.0)
  ICON TERMINAL       = TERMINAL,       SCALE(1.0, 1.0)
  ICON PBX            = PBX,            SCALE(1.0, 1.0)
  ICON SWITCH         = SWITCH,         SCALE(1.0, 1.0)

  INST COMPLETED_CALLS = THERMOMETER_VERTICAL,
                          LOW 0, HIGH 400, INITIAL_VALUE 0, COLOR BLUE
  INST BUSY_CALLS      = THERMOMETER_VERTICAL,
                          LOW 0, HIGH 400, INITIAL_VALUE 0, COLOR BLUE

  LINE PBX_TRUNK_LINE = COLOR LIME_GREEN, THICKNESS 3
  LINE PHONE_LINE     = COLOR LIME_GREEN, THICKNESS 3

  OVERLAY 1 = DRAW_OFFICES IN PHONE BACKGRND
              AT 0,0, SCALE 1, 1, MENU OFFICES
              COLOR BACK_RED

  RTG_EVENT_HANDLER INTERACTIVE_SCENARIO

DATABASE INPUTS
  ASSIGN SFI NEW_DATA.SFI TO READ_SCENARIO_DATA

MODEL SECTION
  SCENARIO_CONTROL
  INSTRUMENT

```

Figure 6-5. GSS Process - PLACE_CALL.

The areas addressed by the Control Specification language are enumerated below.

- Automatic Database Handling
- Interactive Run-Time Graphics
- Interactive Real-Time Control
- Library Controls
- Multi-Simulation Runs
- Optimization

These facilities are built into the control specification using a language that is quite simple, being structured into sections for each of the control statements to be used. A control specification can be six lines or two pages depending upon the number of files and graphical facilities specified for a simulation or task. Most important, the control specification language, along with the resource and process languages ensure that all software built by a modeler or system developer is independent of the platform and OS. All dependencies are taken care of by the language translators. The control specification language is discussed in Chapter 13.

Requirements For Interactive Graphics

As applications become very complex, it is difficult to know what is going on at run-time without a graphical picture. Equally important is the ability to interact with an application graphically. For example, to be able to change the course of the simulation, interactively while it is running, is extremely productive. These needs sparked the concurrent development of the Run-Time Graphics (RTG) system.

RTG supports complex mapping functions, e.g., terrain, bodies of water, foliage, road networks, etc. In addition, large numbers of icons representing moving platforms with radios and sensors can be displayed. Examples of the use of RTG for visualization are illustrated in Figures 6-6a & b respectively. In the early years, Silicon Graphics (SGI) workstations were used to support the complex graphical interfaces, tied to VAX computers running the simulations. Today, the GSS system and the RTG system still run as separate tasks, but typically on the same computer (PCs, Laptops). But they don't have to. In fact, multiple GSS simulations or VSE tasks can be running on separate machines with multiple interactive RTG sessions on the same or different machines.

An Easy To Use Graphics Language

RTG makes it relatively easy for engineers with little programming experience to provide visualizations of dynamic motion in 3D. This is because the complex transformations required to determine the relative positions and orientation angles between icons representing moving platforms are done automatically by the system. To use this facility, the modeler uses special RTG extensions to the resource, process, and control specification languages. These extensions provide for the insertion, update, and removal of icons, lines, and instruments.

They also provide for handling interactive inputs using various panels and buttons. These facilities will be described in a later chapter. The graphics extensions and library facilities for background overlays and foreground plots and diagrams have evolved and grown since the early 1980s. Many new features, such as hierarchical icons, were developed in the mid-1990s to support engineering drawings of the architecture.

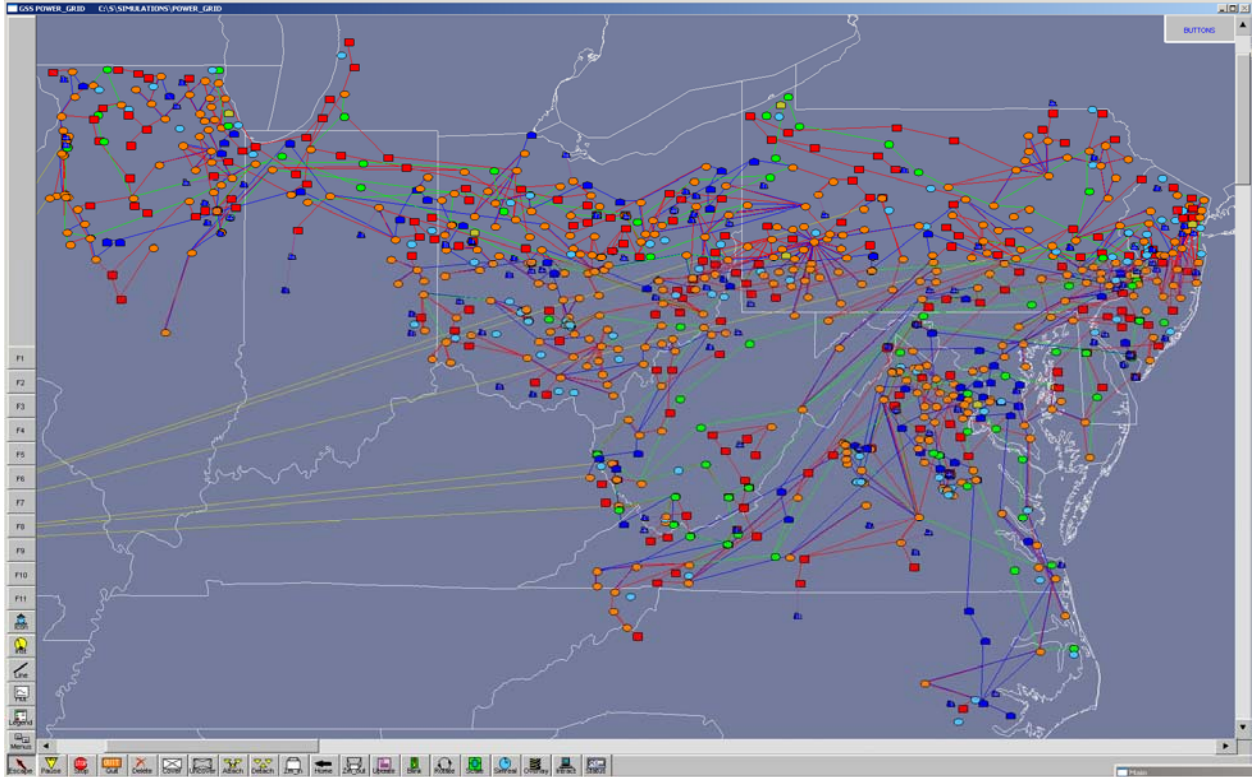


Figure 6-6a. An illustration of RTG graphics.

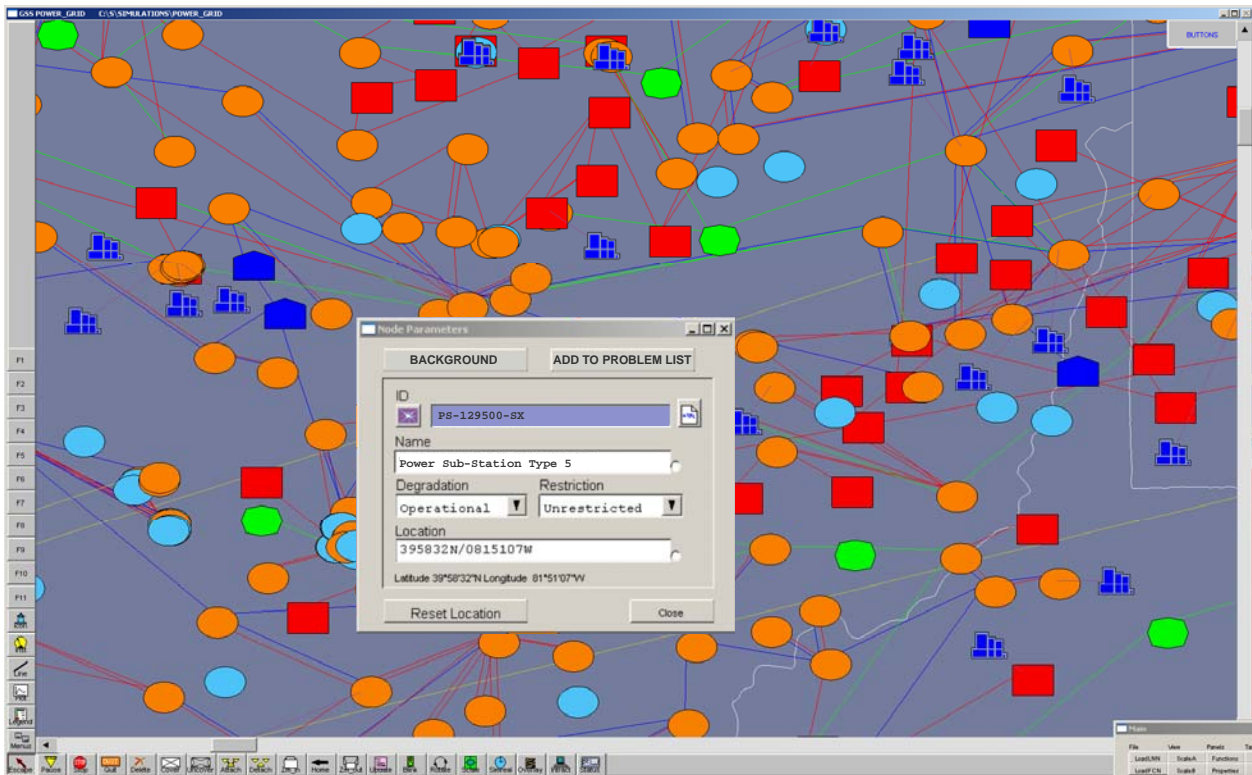


Figure 6-6b. Zoom in, double click on an icon, and up pops a panel.

Scalability - Building The World's Largest Simulation Of A Communication System

As GSS evolved, problems encountered by others, e.g., scaling up complex simulations, disappeared. Using GSS, the numbers of entities and the complexity of the different entity types, appeared to grow without limitation. We attribute this to the properties of *Independence* in model architecture and *Understandability* in both the languages and the architecture. We consider these two properties to be the most important contributors to productivity.

The Software Productivity Paradox

As the client base grew, and larger simulations were developed, additional features and functions were desired - in fact required - to continue the growth of the simulation business. The problem was, the underpinnings of this great simulation environment were built using standard software approaches. As the GSS CAD product grew, putting VSI far ahead in the market, it became impossible to maintain the underlying software, that in which GSS and RTG were written. Product expansion was delayed while many man-months were spent trying to fix existing bugs.

VSI was faced with a software productivity paradox. We had the world's best environment for building complex simulations, but this great simulation environment was built using a classic software environment. Behind the scenes, programmers were faced with the same software nightmares as everyone else.

Solution To The Software Nightmare

In 1990, a decision was made to embark on the development of a second very similar product, the Visual Software Environment (VSE). VSE was also to be used to build itself as well as GSS and RTG. The justification for VSE was simple: rebuild GSS into a software environment that supported the growth of functionality wanted by its users with the same productivity levels afforded for modeling and simulation.

Because VSI's existing GSS system already had everything needed to build general software, one could envision translating the existing code into GSS. However, translating the code was not an option. There was no architecture! And, there was no way to view and assess an architecture for the existing code, even though pain had been taken to organize it into what was thought to be a good approach. It quickly became apparent that, as more functionality was added, the original "architecture" could not hold up under the strain of increasing complexity. More importantly, the system became virtually impossible to change. This is a common software problem that is not readily apparent without being able to visualize and create good architectures. For the first time, an architecture had to be generated for GSS.

The first pieces of software to be rebuilt were the Resource Translator and the Process Translator. As the architectures for these pieces of software evolved, it became apparent that a totally different approach to translation was needed. This new approach involved building and sorting tables in a sequence of independent passes. Many utility modules were created along with managers of the many different databases necessary to support the very complex translation process.

The architecture created using the engineering drawings rendered both VSE and GSS easily upgradeable and supportable. Today, after many upgrades, those architectures are still easy to understand and new features and functions are easy to insert. Most importantly, the architecture itself is easy to change on an incremental basis, i.e., it is easy to take a piece of the architecture and improve it to take on more functionality in a specific area without affecting the rest of the architecture. This is because of the independence of modules, apparent from the visualization of the architecture. In fact, there is effectively one architecture, since the differences between VSE and GSS are almost trivial.

Rebuilding RTG - The Solution To The Visual Development Environment

Having VSE made building software, and reusability of complex modules much easier. This was accomplished without the fully integrated CAD graphical interface shown previously in Figures 6-3 and 6-4. Although third party CAD products were used to produce the engineering drawings prior to 2000, they were never fully integrated into the system. Worse, three different vendor products were used. This is because - one by one - each vendor had the same software problems as GSS, and (all three) went out of the business. Up until the year 2001, the drawings were effectively done off line.

Starting in 1996, RTG was redesigned and rebuilt to support the advanced graphical features needed to implement a fully integrated, interactive CAD drawing environment. To do this properly required the ability to support hierarchical modules that could be disconnected, moved and reconnected differently. It required being able to pop the cover off of an icon, and zoom in to see another layer of hierarchy underneath. This facility is illustrated in Figure 6-7.

To provide a usable hierarchical icon facility, one has to be able to uncover an icon and see the connection lines outside the icon as well as those going into the icon. This requires special transformations to ensure matching the “wires”. The prior version of RTG was designed around a pixel space, forcing the user to implement higher order transformations involving hierarchies of icons. The new version of RTG built all of these transformations into the system, so the modeler or software developer did not have to build and test very complex software to develop sophisticated graphics.

In addition, the new RTG provided for pin connections and pin labels, so that lines could be drawn by users to interconnect icons, while the developer received all the information needed to process the behind-the-scenes database of interconnections. All of this made it relatively easy to build the CAD front end for visualization of the software architecture.

To be independent of the platform and operating system, the code generators produce plain vanilla C and the latest version of Open-GL. All of the graphics are directly transportable among many platforms and operating systems, including all Windows, Linux and Unix platforms

WORKING WITH HIERARCHICAL ICONS

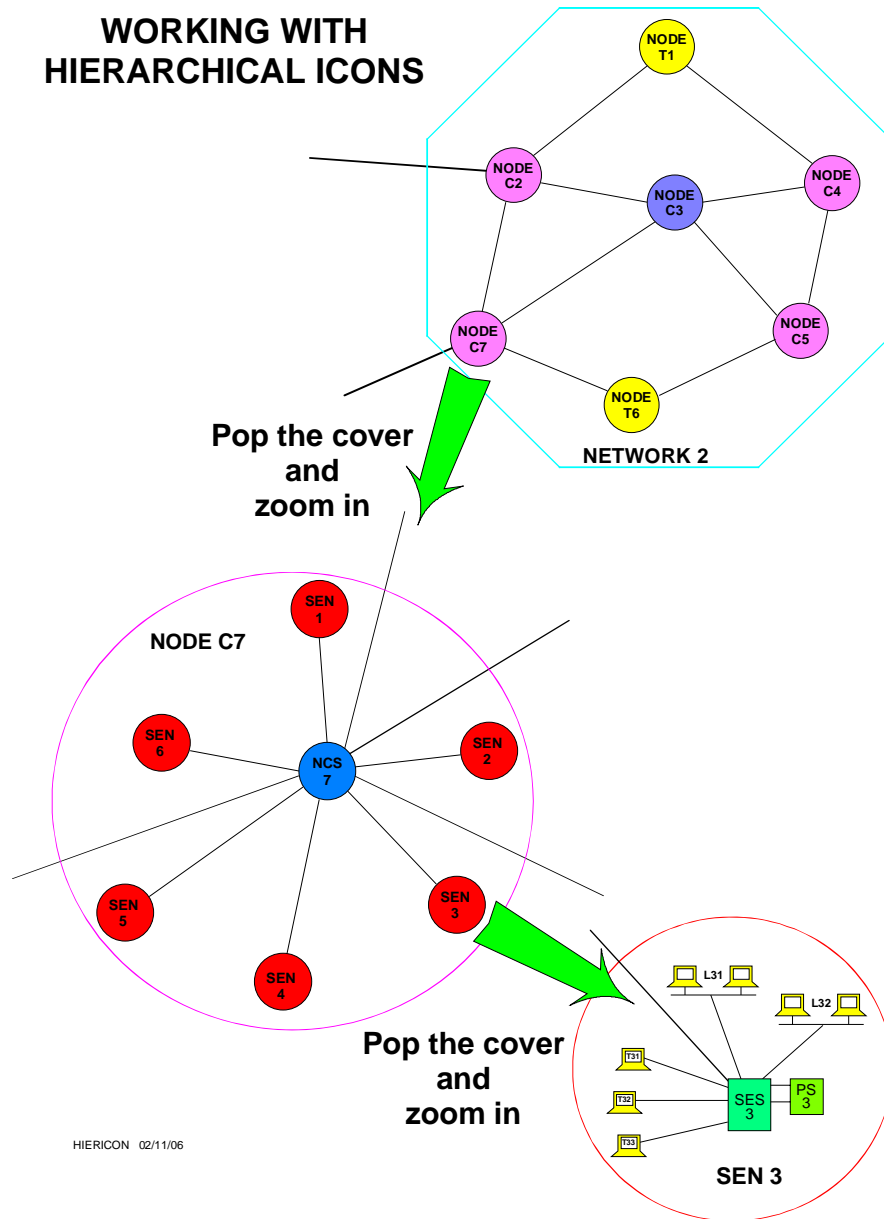


Figure 6-7. An illustration of hierarchical icons in RTG.

Two additional facilities are provided with RTG. One is an Icon Library Manager (ILM). The ILM provides users with the ability to draw complex icons to be used by RTG. These icons are stored in libraries that can be copied and shared among users and applications. They are independent of the platform and operating system.

The second is the Panel Library Manager (PLM). The PLM makes it easy to build panels - the equivalent of X-Windows Motif or Windows dialog boxes - as shown in Figure 6-6b. However, VisiSoft panels are *not* dependent upon the platform or operating system. More importantly, the applications interface is easy to use compared to any of the competitive products.

The Tools To Build A Fully Integrated Visual Development Environment

In 2000, ten years after embarking on the development of VSE, and four years after the new design of RTG, the tools were in place to build a complete Visual development Environment. Note: this was eighteen years after embarking on the initial development of GSS. The directors agreed to finance the development of a prototype to see what it would take, and how it would be received. Because of all the prior experience, this effort came together very fast. By early 2001, an in-house graphical interface for building software as well as simulations was in use.

Figure 6-3 shows the Visual Development Environment (VDE) “window” that is used to build software or simulations. Knowing either GSS or VSE implies knowing the other except for a minor difference. Models built in GSS can be directly transportable into software modules in VSE. Using a GSS simulation for designing and testing software modules provides an ideal environment, requiring no changes to move back and forth to VSE.

Figure 6-4 provides an illustration of two edit sessions open simultaneously. The one on the left is a resource and the one on the right is a process. During a typical session, many edit sessions are open simultaneously, including that of a control specification.

Figure 6-8 illustrates a plot of an engineering drawing that is a small part of a very complex piece of software. Note that all of the change control associated with hardware drawings exists on the software drawing. This includes references to higher level drawings that use the modules in the lower level drawing, module changes and change control authorizations. In addition, by double clicking on a module - at any level - one opens an editor to the documentation for that module.

In addition to modules and tasks are containers. Containers can contain multiple drawings to support export of complete systems from one platform for import to another. When a drawing or container is exported, all of the documentation goes along with the architecture and source code for the resources, processes and control specifications. These can be imported on a different platform with a different operating system (e.g., Windows XP to Linux or Unix), prepared, and running in a few minutes.

BACK ON THE STREET AGAIN

By early 2003, equipped with a totally new product, one that is easily enhanced and refined, VSI was converting all of its simulations into the new environment. In addition, many of the design and corresponding coding rules that could not be enforced without this visual environment were injected. This further improved productivity. No one could build code without first building an architecture. Productivity soared!

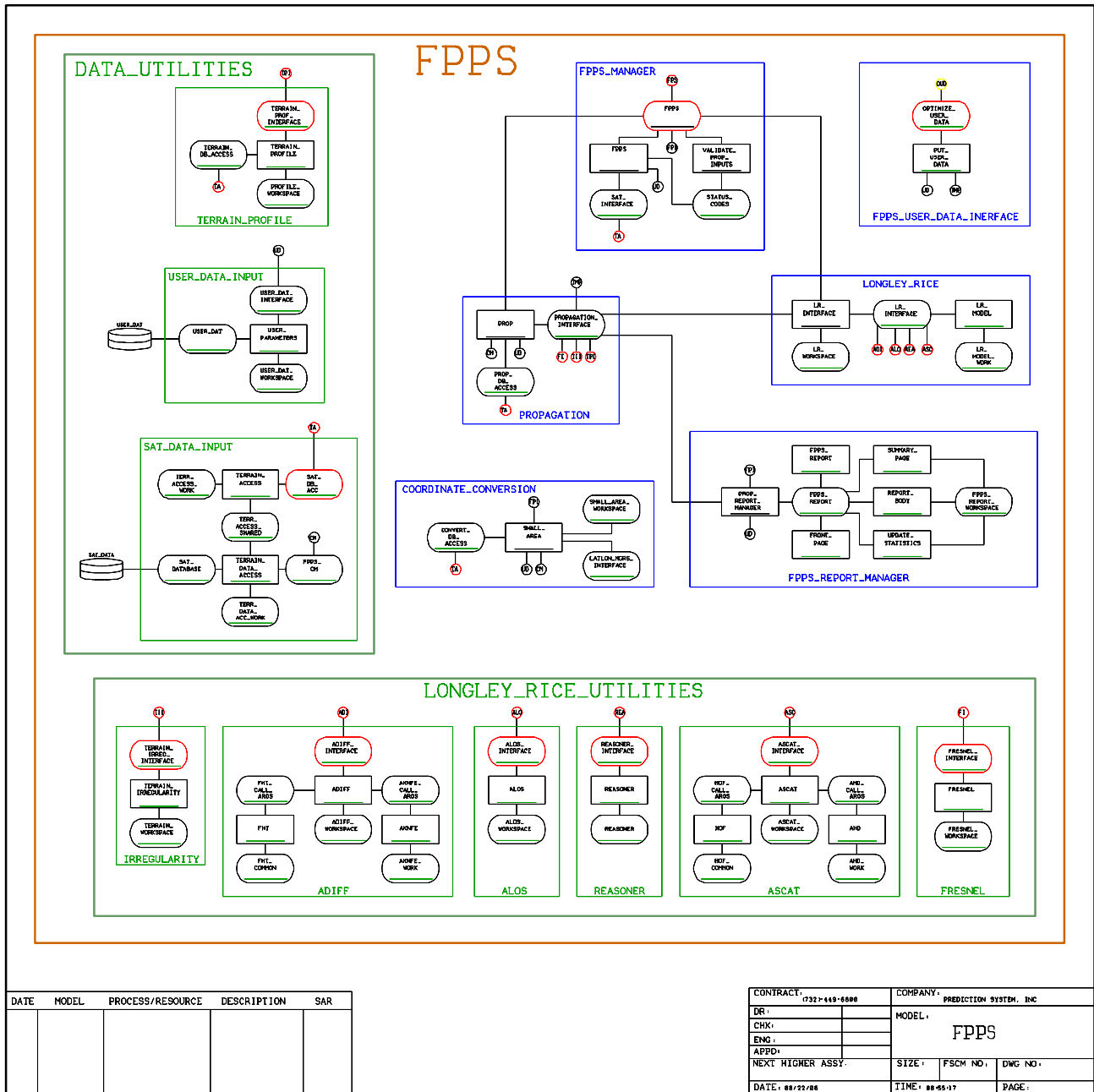
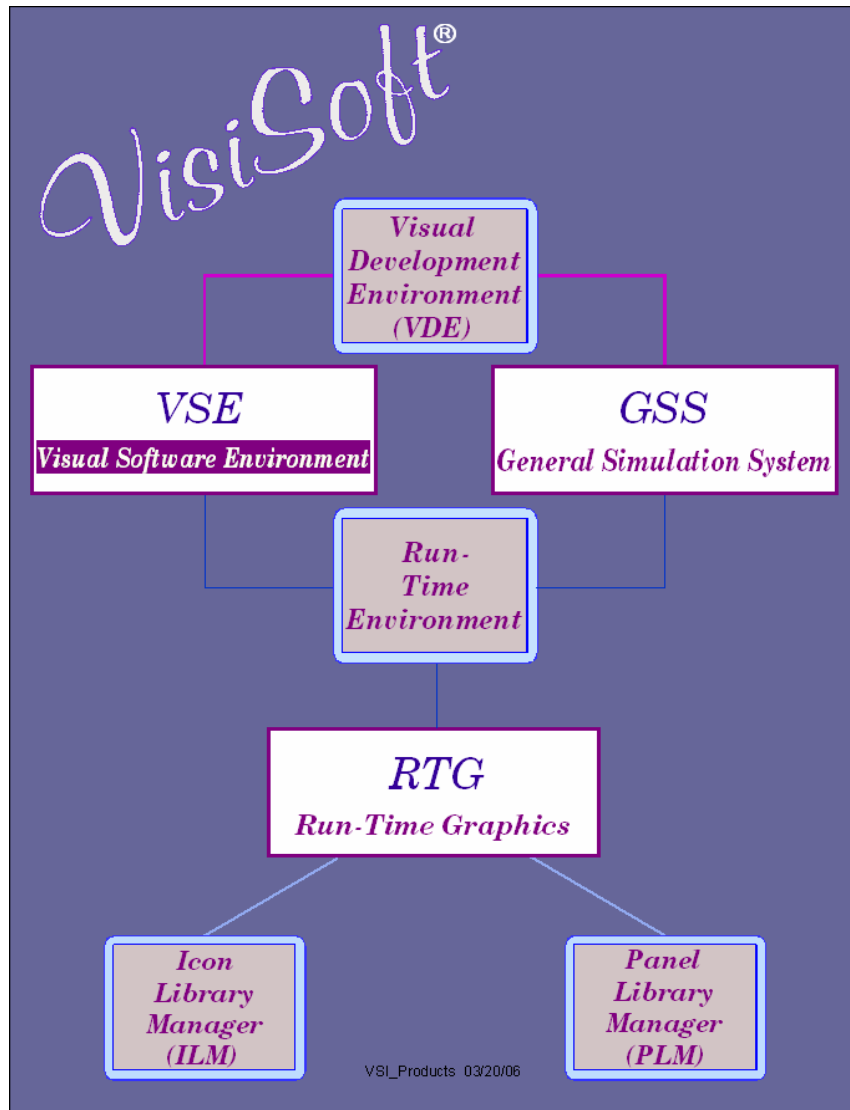


Figure 6-8. An example of an engineering drawing.



Chapter 7. A Technical Overview Of VisiSoft®

As illustrated in the figure above, VisiSoft consists of three systems:

- the *Visual Software Environment (VSE)* for developing and running software
- the *General Simulation System (GSS)*, for developing and running simulations
- the *Run-Time Graphics (RTG)* system that provides a graphical interface for developing and supporting both software and simulation products

These systems are supported by the following subsystems:

- the *Visual Development Environment (VDE)* provides the graphical CAD architectural visualization facility for both *VSE* and *GSS*
- the *Run-Time System (RTS)* that supports software tasks and simulations during run-time
- the *Icon Library Manager (ILM)* for building and managing large libraries of icons to support *RTG*
- the *Panel Library Manager (PLM)* for building and managing large libraries of panels to support *RTG*

Together, these components provide advanced graphical interfaces to develop, support, and interact with complex software and simulation systems. All of these components have been developed and are supported using VisiSoft.

Building Applications

When a developer builds a software product using VisiSoft, an abstracted architecture of that software product can be characterized as shown in Figure 7-1. This illustrates critical facilities available to a developer building and supporting a software product. These facilities are most valuable when the software being produced must run on different platforms. It is these types of facilities (there are others) that make the differences in the hardware and OS platforms transparent to both the developer and the end user.

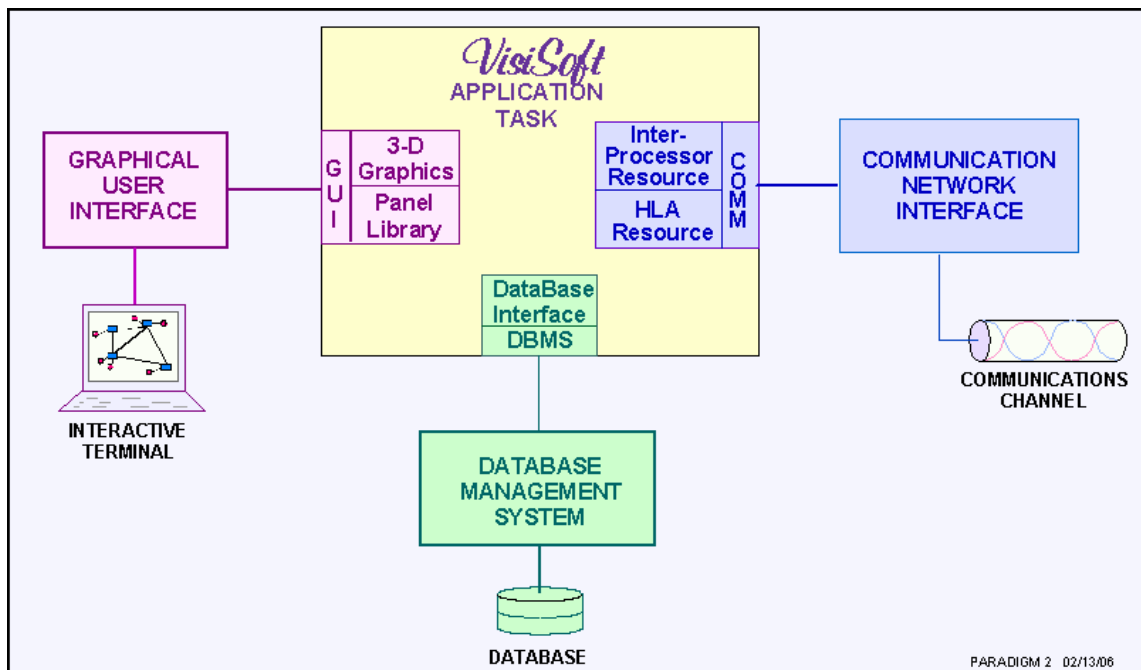


Figure 7-1. An abstracted software architecture.

The resulting layers of insulation from hardware and OS platform differences are illustrated from a different perspective in Figure 7-2. With these facilities, end users can work with multiple interactive graphical workstations, databases, and communications networks without changing the software. From a productivity perspective, the developer is also insulated from platform differences.

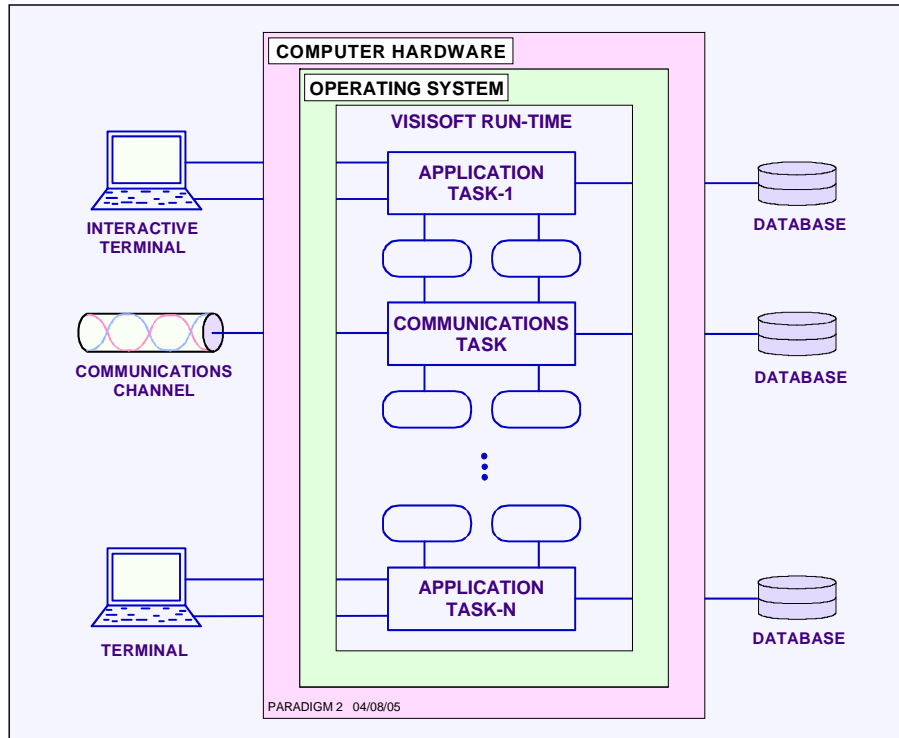


Figure 7-2. Layers of insulation from hardware and OS differences.

Multi-Tasking On Multiple Processors

Taking a slightly deeper look, products can be developed that use multiple platforms simultaneously. The only constraint is that these platforms are connected using IP network interfaces, e.g., the Internet. Interprocessor resources are used to automatically handle the communications channel protocols. Developers simply put data into and take data out of these interprocessor resources while issuing channel read and write statements.

Applications on a single processor may contain multiple tasks that are running concurrently. (These are equivalent to UNIX processes.) Tasks can communicate by sharing memory directly. This is the fastest way to communicate, and in VisiSoft it is easy. One simply uses Intertask resources, moving data in and out directly. Using the facilities described above, one can feel comfortable writing applications that use multiple tasks on multiple processors with a day or two of training.

Facilities exist for running large scale discrete event simulations very efficiently on multiple processors. These facilities include provisions for automatic cross-processor scheduling, time-synchronization, and interprocessor resource coherency management. The scheduling and memory management software running behind the scenes alleviates the modeler from any concerns about otherwise formidable software problems. Developers can focus on solving end user application problems.

ARCHITECTURAL FACILITIES

In the next two Chapters we will be describing the VisiSoft architectural features and approaches to building architectures that support reusability and scalability. An overview of these facilities is provided in Figure 7-3 below.

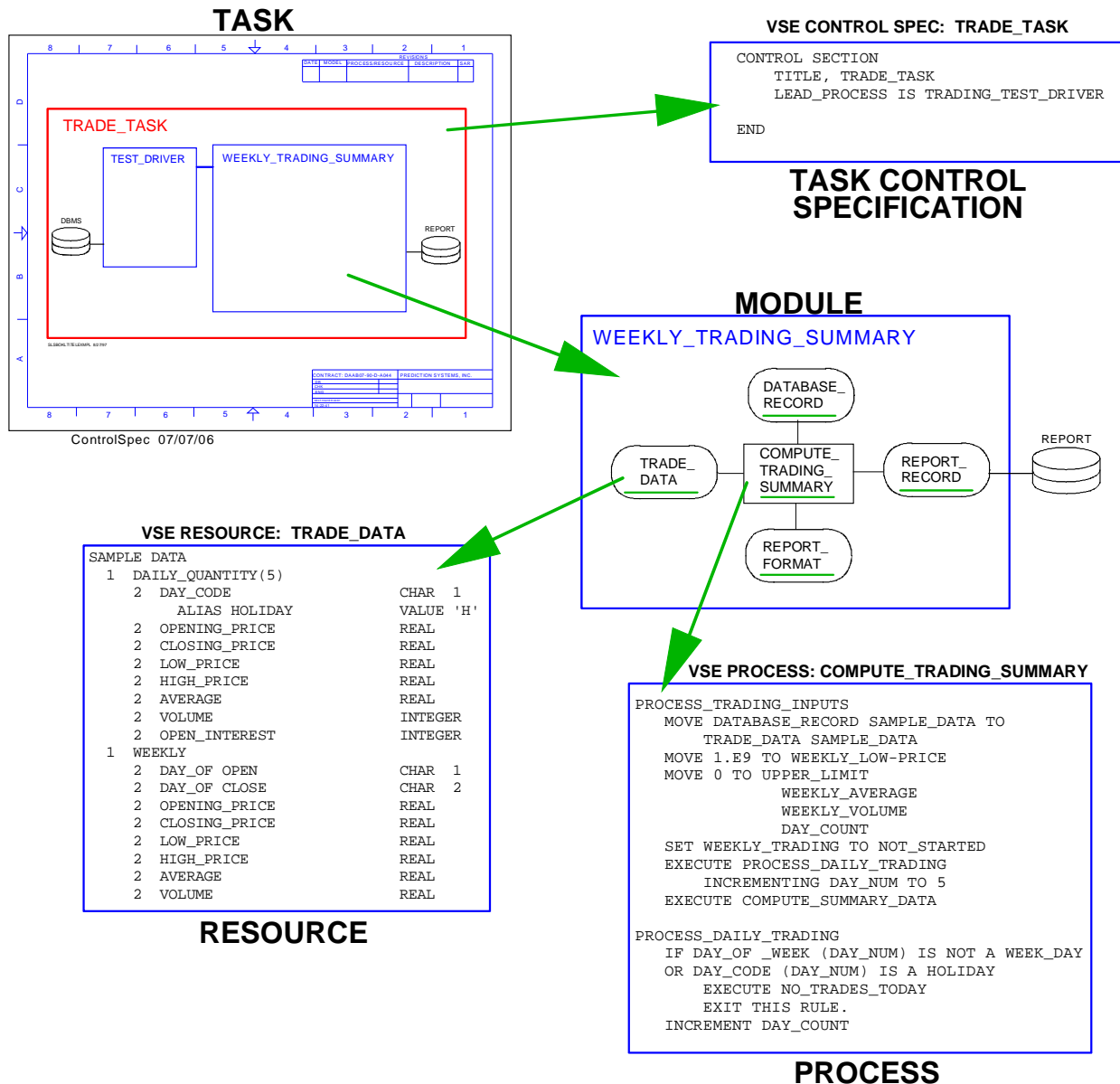


Figure 7-3. TASK architecture illustration.

Tasks/Simulations

Using VisiSoft, applications are broken into tasks. Tasks are defined by Task Control Specifications and the modules that comprise the task. In the case of a simulation, the Simulation Control Specification defines the simulation task that is comprised of simulation models and software modules. Tasks are represented by a red border on the drawings.

Modules/Models

Referring to Figure 7-4, software modules (simulation models) are of two types, elementary and hierarchical. Elementary modules contain resources and processes. Hierarchical modules can contain both elementary modules and other hierarchical modules. Modules are represented by a blue border on the drawings.

Utilities

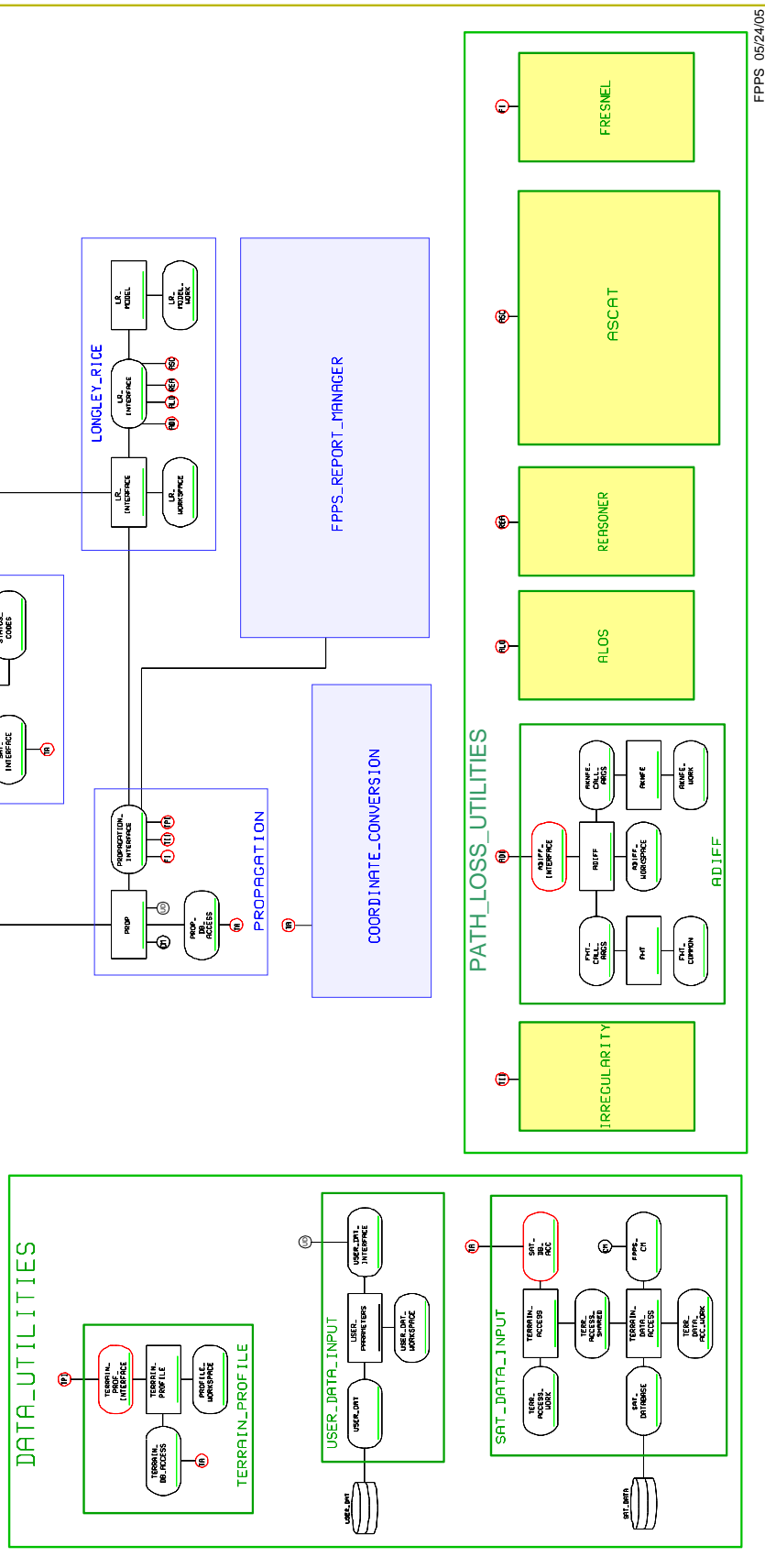
Utilities are software modules that are used by two or more processes contained in one or more separate software modules. Utilities may appear in more than one drawing in the same user directory. They must be connected to the modules that use them using a connector. Although the architecture and source code may appear and be changed in more than one drawing in a user directory, there is only one instance in that directory. Utility changes in one drawing automatically appear in the other drawings in that directory. Utilities are represented by a green border on the drawings as shown in Figure 7-4.

Library Modules

Library modules are utilities that are shared by different user directories across different platforms. The drawing of a library only appears in the user directory where the library is maintained, providing a high degree of protection from unwanted modification. Library modules must share data with the modules that use them using *aliased* resources with connectors. Aliased resources are templates for data shared by the calling process. The using module must have *shared as* resources that provide the data used by the aliased resources in the library module.

Libraries may contain multiple library modules inside library archive files. These modules are automatically added to the library archive file when the library module is prepared. Library modules for the same library may reside in different directories as long as the most recent version of the library archive file resides in the directory in which the library module is being prepared. Libraries are represented by a dark gold border on the drawings.

PROPAGATION_PREDICTION



FPPS 05/24/05

CONTRACT:	COMPANY:
DIR:	MODEL:
CHK:	PROPAGATION_PREDICTION
ENG:	
APPD:	SIZE:
DATE: 05-01-04	TIME: 11:08:18
	FSCM NO:
	DWG NO:
	PRGE:

Figure 7-4. Illustration of a software engineering drawing.

DATE	MODEL	PROCESS/RESOURCE	DESCRIPTION	SPR

Engineering Models

To ensure that user requirements are met, engineers build mock-ups or models. These models can contain simulated controls and meters or displays, so that the user can "try out" the user interfaces in a simulated environment to insure that what is going to be built is what is wanted. Computer simulation is used heavily in the engineering field, with detailed models being used to check out designs before one commits to costly fabrication and testing processes.

In the case of communication system design, e.g., design of switches, digital radios, routers, etc., complex algorithms are embedded in simulations of their environments so their designs can be tested and optimized before they are implemented in software. But, there is no reason to rewrite these algorithms in a programming language. They can be directly translated from models in the simulation to final software code, reference Maslo [64]. If there is a problem with the algorithm, the situation can be replicated in a simulated environment, fixed, and tested. The new code can then be regenerated automatically for the production environment.

Engineering Drawings -- Of Software

An illustration of a wave propagation model (a relatively small drawing by VisiSoft standards) is provided in Figure 7-4. This is a library module, depicted by the dark gold color of the boundary. It also contains utility modules (green boundaries) as well as standard modules (blue boundaries), all visibly identified by their color.

After some experience with this system, one can look at a drawing and recognize the architectures used for different modules. This is because there are many standard architectures used to support categories of functions. As one evolves from the world of programming in a language into design of the software using engineering drawings, one quickly recognizes that:

the design of the architecture is much more important than the code.

This is true from both an understandability and independence standpoint. These concepts, and the corresponding approach that has evolved for building software architectures, will be covered in later chapters and supported with examples.

Engineering Specifications And Documentation

To support the continued enhancement and refinement of a system design, one wants to understand the underlying rationale as design decisions are made. This is typically recorded in engineering notebooks or other external documents. These documents normally contain different design considerations and comparisons, details of trial designs, laboratory test results, field test results, theoretical or comparative references, etc. Complex user interfaces can be specified in detail by writing a user's manual so the end user can review and envision what the system will do. User's manuals can be developed with the aid of an interactive simulation, using models of what the user will have as an interface. Screen shots can be taken from a running simulation of the proposed system.

LANGUAGE FEATURES

In subsequent chapters we will describe the language features of VisiSoft. In general, one cannot write code without building an architecture. It is uncommon to give less senior people the responsibility to create or even change architectures. It typically takes a much higher degree of experience to create or change an architecture. Once an architecture exists, code can be entered by double clicking on the element of interest to open the editor. When the editor is shut down, the code is saved and the management system updates the state of the element. Colored bars inside resources and processes indicate whether the element is prepared (green), in error (red), or changed but prepared (black).

Resource Language

The resource language is tailored to describing data as hierarchical structures, for example, those used to represent messages sent to or received from communications channels, records that are read from or written to databases, and internal memory shared between processes. Large multi-dimensional tables can be built that contain complex data structures.

An important feature is “What You See Is What You Get” in memory. This allows for large data blocks to be moved as a character string into memory blocks that are defined by detailed data structures. There is no “word boundary alignment” performed. Numbers may appear anywhere in a data structure. This provides for fast movement of large complex data structures.

The resource language is designed to support a process language that makes it easy for humans to write easily understandable code. This implies data types that make conditional statements easy to read. It also inhibits testing and movement of data at run time that do not fit specific requirements. These are checked at time of translation, significantly reducing the probability of bugs.

There is no global data. All resources are accessed by pointer (automatically).

Process Language

The process language is designed to maximize understandability. This is particularly true for conditional statements. Anyone familiar with the application being implemented, should be able to understand algorithms that are logically complex without the added burden of learning another language. This property is particularly important in modeling and simulation where validation of models is a critical aspect of a project, and typically requires engineering personnel not experienced in programming to review the code.

Having a readable language makes it much easier to understand the logic and avoid logical errors that are generally difficult to find. It provides the important benefit of allowing subject area experts write code directly.

Control Specification Language

The control specification language is designed to support a highly structured specification that contains lists of specified elements, e.g., icons, lines, instruments, databases, etc. It provides for enumerating the properties of these elements, e.g., colors, numbers, etc. It is a very simple language that eliminates the need for scripts or JCL to run a complex task.

GRAPHICAL FEATURES

Built-in graphical facilities remove the burden of writing complex graphics code. One need not learn a graphics language to build superior 2D and 3D graphics. The Run-Time Graphics (RTG) system provides additional statements that go into resources, processes, and control specifications. These statements control the various graphical elements available to a user. These elements include panels, icons, lines, instruments, plots, legends, etc.

Because panels and icons are potentially complex graphical elements, they are created and maintained using the Panel Library Manager (PLM) and the Icon Library Manager (ILM). These managers provide drawing boards for the user to graphically create complex panels and icons with ease. The libraries of elements can be shared easily between users on different platforms or operating systems using built-in export and import facilities. A large number of graphics libraries already exist to provide support for most graphics applications.

RTG supports complex geo-physical mapping functions, e.g., terrain, bodies of water, foliage, road networks, etc. In addition, large numbers of icons representing moving platforms with radios and sensors can be displayed. Examples of the RTG window for 2D and 3D are illustrated in Figures 7-5a & b respectively. In the early years, Silicon Graphics (SGI) workstations were used to support the complex graphical interfaces, tied to VAX computers running the simulations. Today, the GSS system and the RTG system still run as separate tasks, but typically on the same computer (e.g., PCs and Laptops). But they don't have to. In fact, multiple GSS simulations or VSE tasks can be running on separate machines with multiple interactive RTG sessions on the same or different machines.

Software for graphical depiction of motion, particularly in 3D, can be extremely complex. Moving platforms, e.g., aircraft, require six degrees of freedom to define their position and orientation. When communicating, the position and direction of antennas on platforms are required to determine an accurate estimate of the antenna gains between transmitter and receiver. Being able to see the relative orientations of multiple platforms provides a rapid means for verifying and validating complex models.

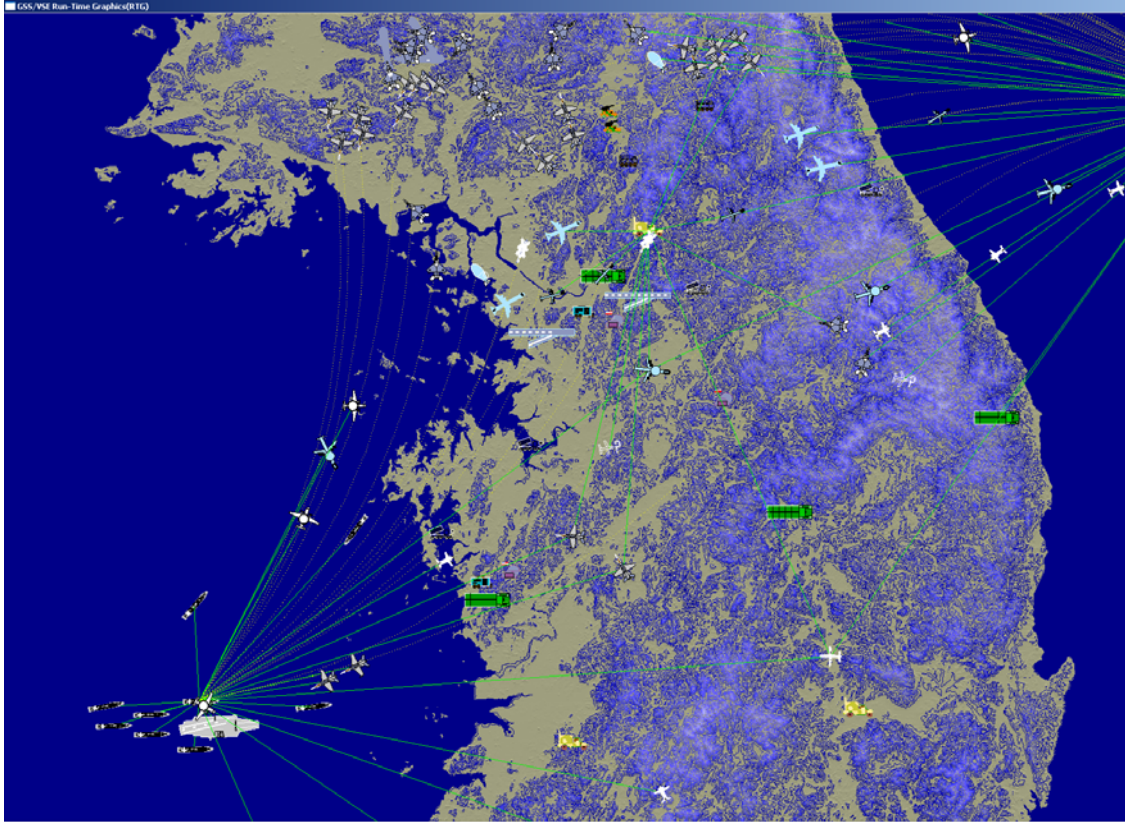


Figure 7-5a. An illustration of the RTG graphics window in 2D.

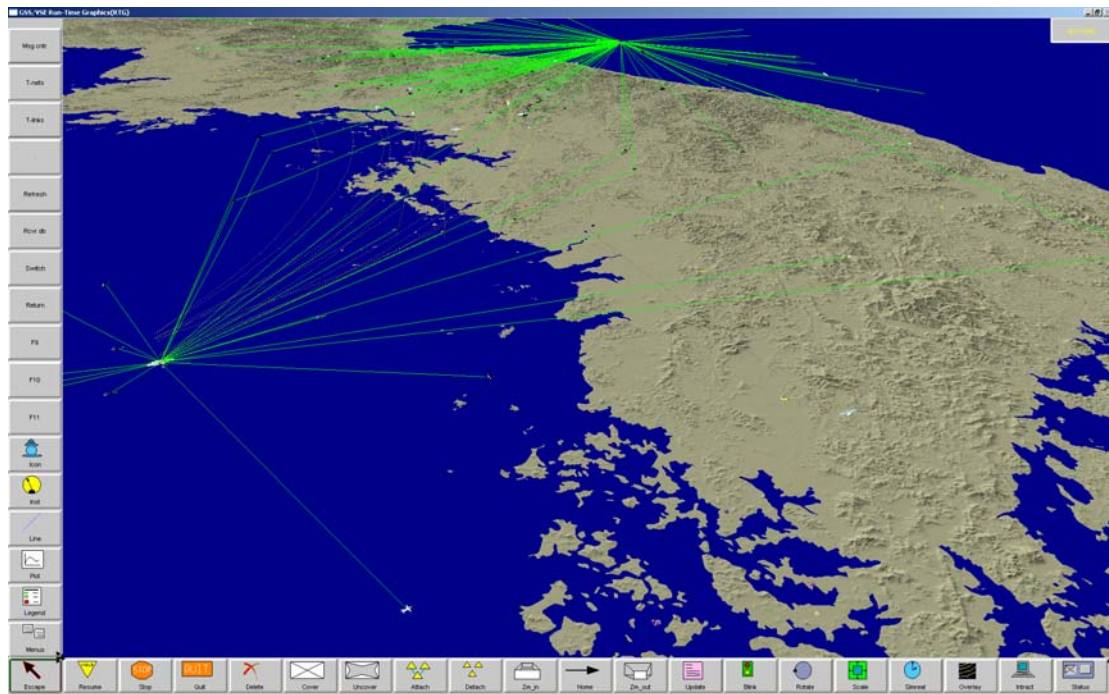


Figure 7-5b. An illustration of the RTG graphics window in 3D.

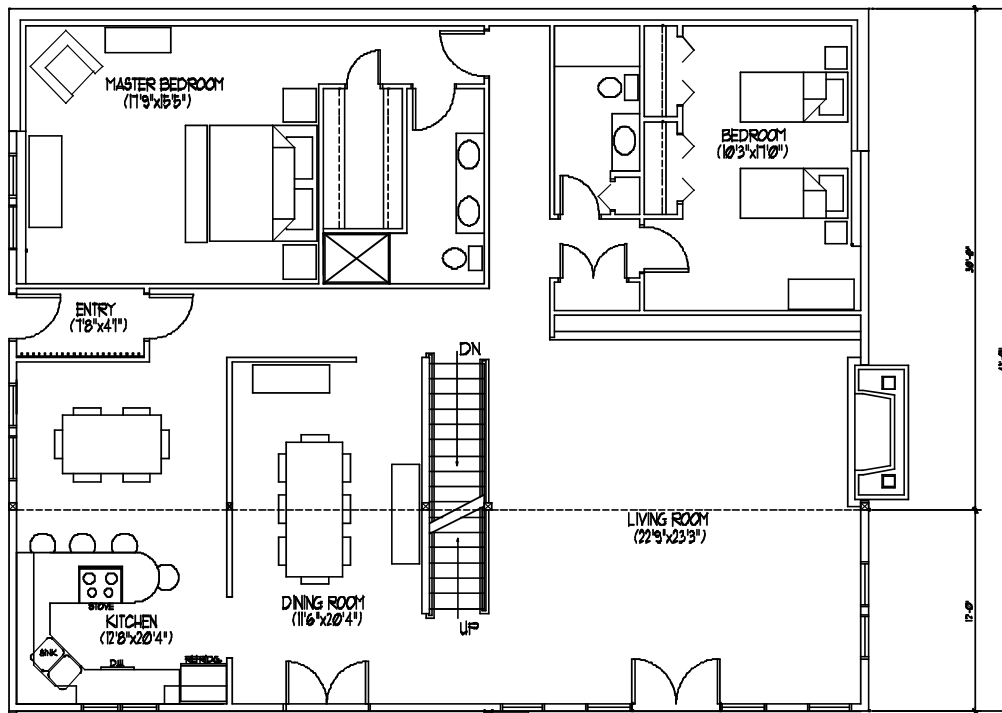
Constrained Nonlinear Optimization

A significant feature of GSS is the built-in optimization facilities used to design complex algorithms or system parameters, and optimize model parameters to maximize prediction accuracy. The optimization methods have been derived from those used for design of highly nonlinear systems. It provides for nonlinear constraints, both inequality and equality, as well as nonlinear optimization criteria.

To use the optimization facility, one need not formulate the problem mathematically. One only has to provide numbers for the constraint values and the optimization criteria, as well as ranges on the unknown parameters. Models can contain decision processes based upon non-numeric states as well as mathematical models. Solutions do not depend upon model formulation.

Parallel processing

With the flattening of Moore's curve, and no recuperation in sight, the only way to achieve faster run times is to resort to parallel processors running together to speed up a single task. However, except for very special problems, parallel processing has been practically unusable ever since it was first considered. VisiSoft has the facilities to make it just as easy to build models and modules to run efficiently on parallel processors as it is to build them for a single processor. If the architecture is designed using good standards for producing independent model instances, the number of processors used should not change the architecture or code.



Chapter 8. Software Architecture

DEVELOPING ARCHITECTURES

Most readers will relate to the above drawing. As in other fields, architecture is much more graphical than algebraic or textual. Whether designing machines, ships, buildings, or computers, architects produce drawings. These drawings are not “approximate” or just suggestive, but rather precise engineering specifications that are used directly in production.

To facilitate the graphical approach, CAD tools are used extensively. The time to produce and reproduce drawings has been cut dramatically since the days of drawing each line by hand. Reuse of drawing parts is common. They are copied and modified easily.

But today these drawings can be converted to XML, a representation suitable for computer processing. An experienced XML programmer can create an architecture and make changes without ever looking at the drawing. Is the use of drawings a legacy approach?

In the business environment of architects, eliminating the use of drawings would be ridiculous. One would not consider creating or even changing a drawing in a language like XML. XML is a standard for storing and exchanging files; it is of no use in design.

Now let’s consider software. Drawing tools are not used for designing software architectures. Instead, Programmers pride themselves in their ability to understand esoteric languages, and to create and maintain software directly in these languages. More importantly, until *VisiSoft*, there have not been any drawing tools available that precisely represent a software design. Having used *VisiSoft*, the above architect example is not an absurd parallel.

ARCHITECTURE - A NEW SOFTWARE CONCEPT

As described in the prior chapters, VisiSoft is a Computer-Aided Design (CAD) tool that provides a precise visualization of the architecture of a software system. The engineering approach provides a one-to-one mapping from the top level architecture to the code. Using VisiSoft's graphical CAD front-end, one can drill down from the top system level drawing to the details of the code, with no abstractions in between. Interconnections are as meaningful at all levels of a drawing as they are in electronic circuit design, logical design, or machine design. The VisiSoft CAD environment is derived from the same concepts used by chip manufacturers for designing hardware.

Using VisiSoft, decomposition of the architecture and composition of the detailed design is accomplished using graphical symbols that directly represent the software. With this approach, it quickly becomes apparent that architecture is the most important part of software design. Having used this system, one could not imagine working without drawings. One also observes that software architecture is meaningless without the totally new paradigms that VisiSoft provides. In software design courses using VisiSoft, architecture is taught first - before language or coding facilities are described. With this approach it becomes clear that architecture has the most effect on productivity, especially in the support phase of a product.

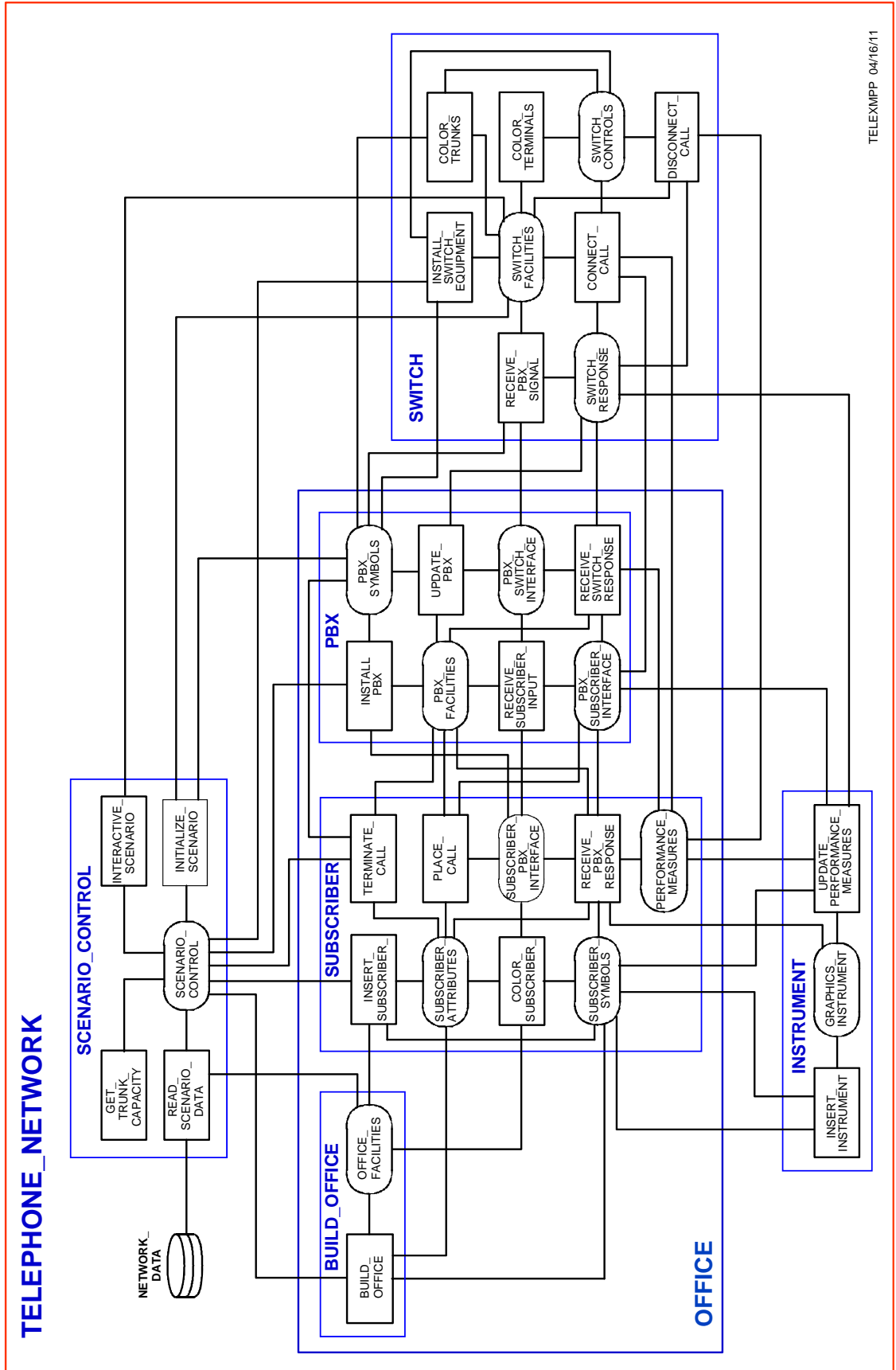
USE OF ENGINEERING DRAWINGS

Although written documentation is important, in practice, engineering drawings are the essential tools to support the planning, review, and assessment process needed to control a project. This is because of the requirement to develop and modify the structure of modules as the design unfolds.

Engineering drawings provide the means for creating and improving the structure of software. This is because given the proper CAD tools, these drawings can be modified easily to implement structural improvements. Also, the best structure for a complex set of modules cannot be known until most of the design has been completed and carefully reviewed with the software team. Only after understanding all of the facilities that must be built into a module, and how those facilities interact, can the developers decide on the best architecture for a module. This implies that a module may be built initially using an inadequate structure before one can see how to improve that structure.

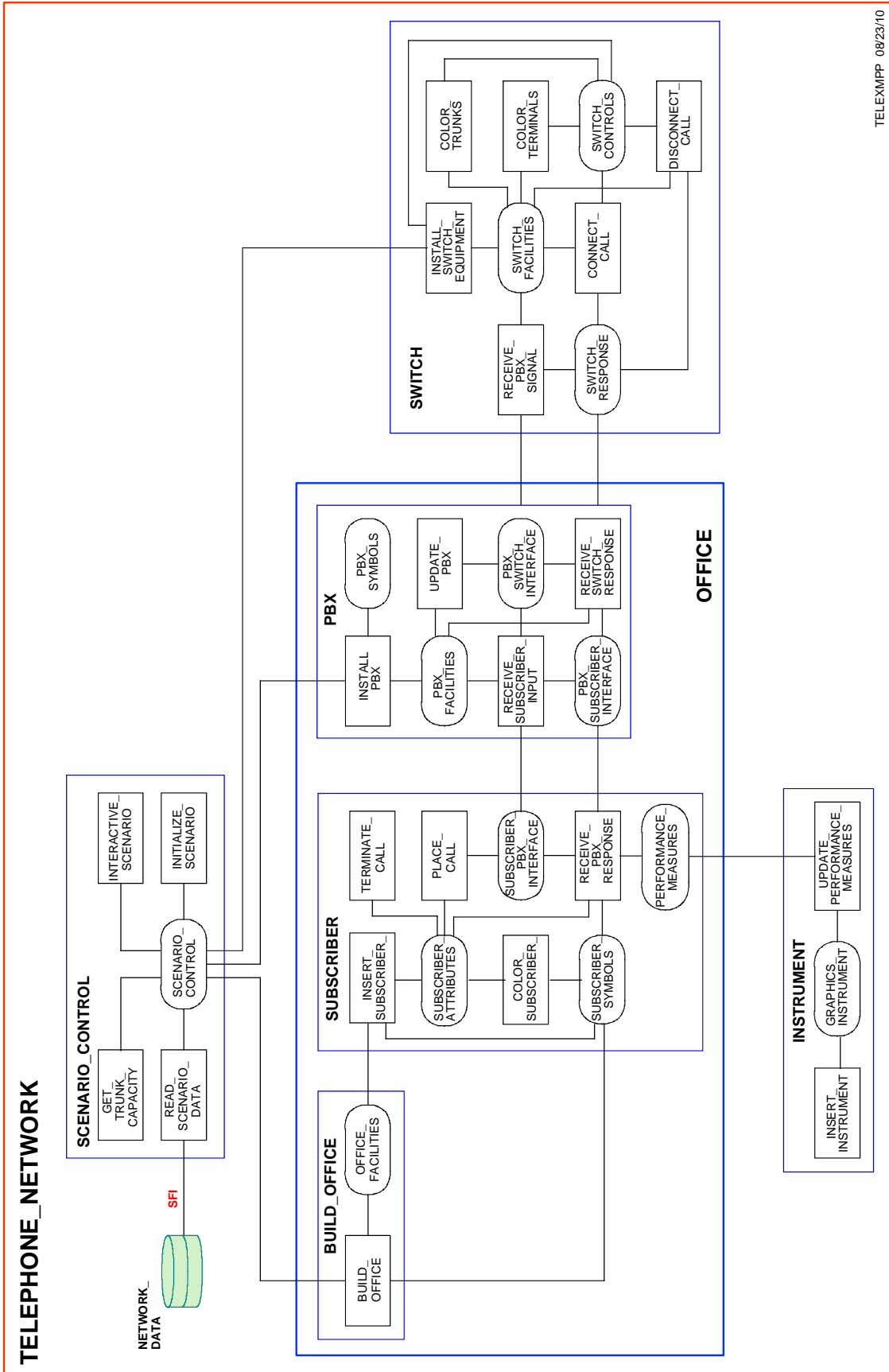
Visualization Of The Properties Of Independence

Geometry and algebra each play important roles in engineering. Theoretically, one could do away with the images provided by geometry. In practice, those who can use geometry have a significant advantage for many problems. Figure 8-2 illustrates a model developed without engineering drawings. As is typical in conventional software, data is shared everywhere, with no visible check on independence. Figure 8-3 is the same application developed using VisiSoft. Needless to say which one is easier to understand and change - thanks to the independence properties.



TELEXMPP 04/16/11

Figure 8-2. Telephone network model - a pre-drawing version.



TELEXMPP 08/23/10

Figure 8-3. INTER_OFFICE_NETWORK Model using RTG.

SOFTWARE ARCHITECTURE

An overview of the VisiSoft approach to software architecture is provided below. Just as with any other architecture, one must account for all of the facets of the problem. Software architectures must support the user environment as well as the development and support environment. Although most of our focus is on development and support, the user environments will be discussed where appropriate.

We start by defining the components of a physical architecture. Refer to Figure 8-4.

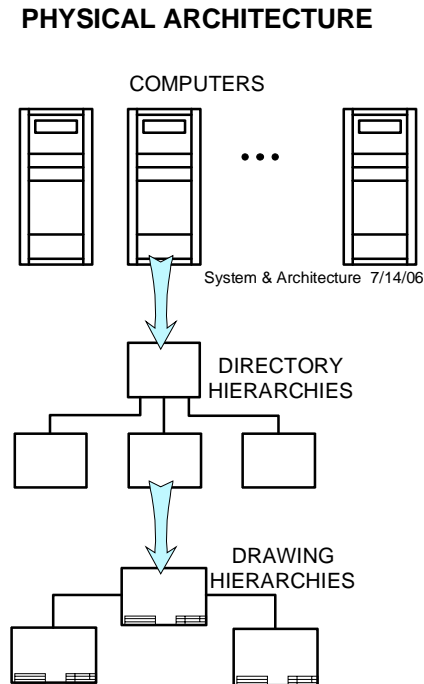


Figure 8-4. VisiSoft view of the physical architectural components.

Run-Time Software Systems

Using VisiSoft, a run-time software system may span one or more computers, typically using communication links when more than one computer is used. A system may span different platforms and operating systems.

On a given computer, multiple executable *tasks* may reside in one or more directories. When one task starts another task, the second task becomes part of an executable *task hierarchy*. When tasks are started by separate user actions, they are *independent*. During execution, tasks that reside on the same computer may attach to multiple *shared memory segments* as part of a task hierarchy. Independent tasks attach to *global memory segments*. Because of the ease with which multiple tasks can share memory directly in VisiSoft, *threads* (p-threads) are used only in a parallel processor environment running under a single OS.

Development Environment Components

Multiple computers are typically used during a software development. System components may be *exported* from one platform (e.g., Linux) and *imported* to another platform (e.g., Windows). From a developers perspective, the development environment is the same on every platform. Each platform contains the following facilities:

- **VisiSoft User (Developer) Directories** - On a given computer, one can assign multiple VisiSoft *User Directories* to house one or all of the components of a system. Different versions of the same component may reside in different user directories.
- **Drawings** - One or more drawings may be contained in a user directory. A drawing may contain a single task or module. Each of these may contain one or more modules in a hierarchy. To create or modify components, one must create a new drawing or modify an existing drawing. Drawings may be deleted without deleting the components in that drawing. Components may be deleted from a drawing or from the user directory, in which case they are deleted from all drawings.
- **Containers** - Containers are used to export and import drawings, so multiple drawings may exist in a container. Drawings cannot be modified in a container. It is common practice to store all of the drawings in a user directory in one or more containers. A directory can be backed up completely at any time simply by exporting one or two containers.

ARCHITECTURAL COMPONENTS

The basic architectural components of a software system are shown in Figure 8-5. They have been introduced in prior chapters. These are resources, processes, modules, and tasks. Their architectural properties are described below.

Tasks

Tasks are executable modules at run-time. The VisiSoft logical system hierarchy illustrated in Figure 8-5 contains 4 tasks. Architecturally, a task can start one or more tasks, and a task can invoke one or more modules by starting a process. Modules within a task drawing may be elementary or hierarchical. There is no limit on the hierarchy. However, a task containing 8 levels of hierarchy is a huge piece of software (on the order of 1M lines of code). The modules in a task need not reside in the task drawing.

Hierarchical Modules

Figure 8-6 contains an illustration of a relatively complex module hierarchy. At the bottom of the hierarchy, Drawing Level 1, are elementary modules. Hierarchical modules are illustrated in Drawing Levels 2 and 3. It is not unusual for complex systems to take up to nine or ten levels of drawings. At this level of complexity, one can expect to be over 1M lines of code.

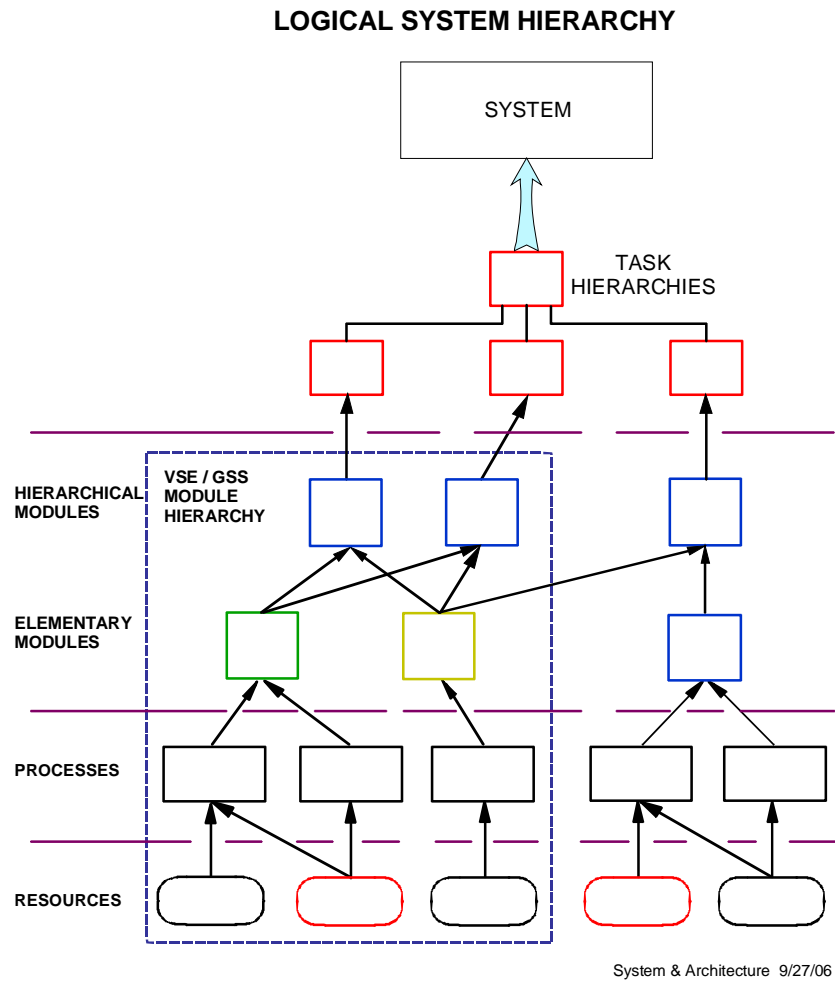


Figure 8-5. Overview of the VisiSoft logical system hierarchy.

Elementary Modules

The module hierarchy in Figure 8-6 is apparent, down to the elementary modules that contain resources (ovals - representing data structures) and processes (rectangles - representing rule structures). Connections are designed at the elementary level to maximize independence between hierarchical modules. This allows true reuse of modules.

System Decomposition And Module Composition

The decomposition of a system into a hierarchy of modules takes considerable experience in the particular application being developed as well as in software architecture. We note that, from a VisiSoft standpoint, complex applications include language translators and operating systems. The grouping of resources and processes into elementary modules is an important architectural design function. All resources and processes must lie within an elementary module boundary. This implies selection of the module that will contain the interface resource between two modules. Those responsible for that module have implicit control over that interface.

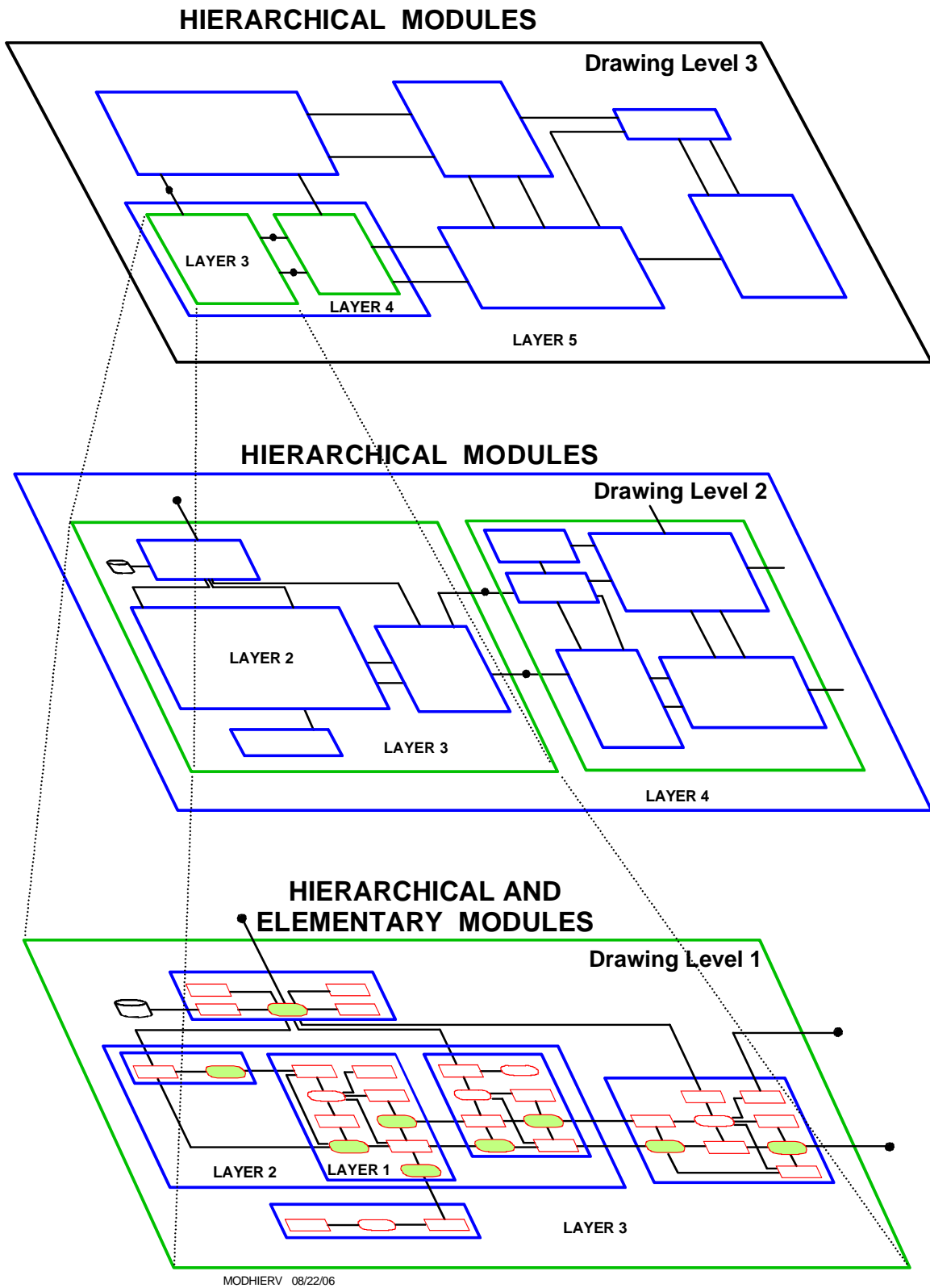


Figure 8-6. Illustration of a VisiSoft model hierarchy.

Most importantly, architectures need not be poured into concrete. On the contrary, using VisiSoft, they are easy to change. This is because the processes, resources and connection lines are moved easily from one module to another. So if one decides to move an interface resource from one module to another, it is a simple but very visible drawing change that confers control of the interface to the other module.

Categories Of Modules

There are three categories of modules in VisiSoft. These categories provide different levels of protection with regard to change. Both elementary and hierarchical modules can reside within each category. Modules of any category can only appear once in a drawing. The rules for these categories are described below with examples in Figure 8-7.

- **Modules** - have a blue border. These are the basic building blocks in a task. In VisiSoft, modules can be decomposed hierarchically, i.e., they can contain submodules and sub-submodules, etc. Modules can only appear in a single drawing in a user directory, and are meant to be unique, i.e., not reused, across directories.
- **Utility Modules** - have a green border. These are modules that are reused by processes in the same directory, and can appear in more than one drawing. They are typically used to manage separate databases or perform utility type functions. The green color flags them for change protection. If they are changed to accommodate a given process, that change must be compatible with the other processes that use them.
- **Library Modules** - have a gold border. These are utility modules that can be shared from different directories and different computers. They are stored as object modules in an object library file. The source only appears in the directory where they are maintained. Processes in a library module are called from an application using their process name, module name, and library name. Since each of these names must be unique within the next level of hierarchy, there can be no duplicate names when linking to library modules in VisiSoft.

The functions of a VisiSoft library module can be upgraded while at the same time preserving the original module in the library for prior users. Users can call the new function using the same process name within the same library by using the new module name. VisiSoft has a large set of libraries that support various applications, including 3D graphics, that are shared easily.

VisiSoft libraries have been designed to be controlled separately under special protection mechanisms. But given access to a library directory, the responsible person sees everything that is needed to allow for ease of changes and testing. Library directories typically contain regression test drivers and data sets to ensure changes meet all prior, as well as new requirements.

The top level module in Figure 8-7 is a library module. It contains modules and utility modules that are immediately identified by the color of their border. Libraries may contain a virtually unlimited number of modules, and individual library modules can be huge hierarchies. The library module shown in Figure 8-7 performs electromagnetic wave loss propagation predictions between two antennas using detailed terrain, foliage, and building data. An example of a medium size task architecture is shown in Figure 8-8. Being a model of a communication system, this architecture has some special properties that will be described in Chapter 9.

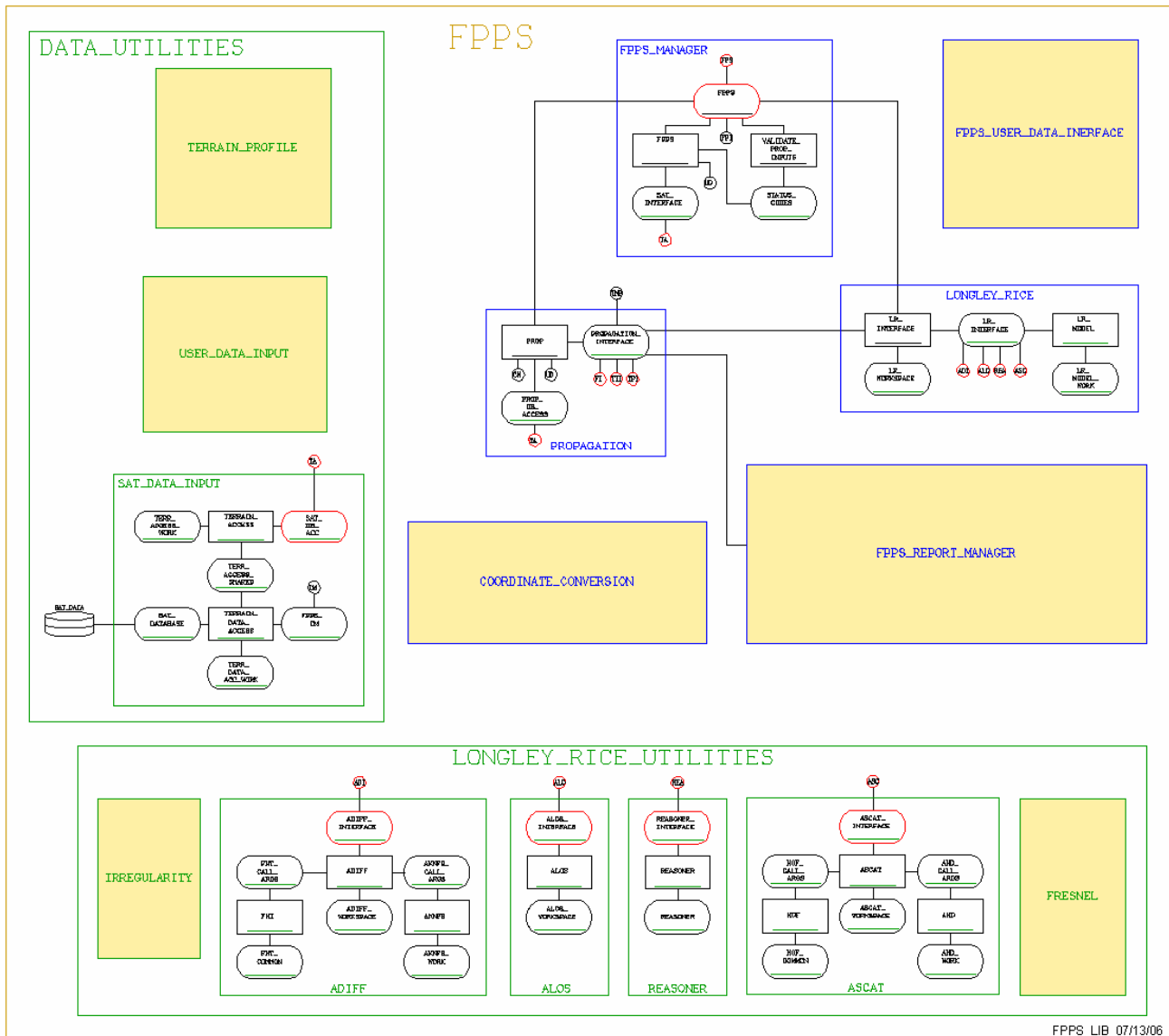


Figure 8-7. Example of a library module with utility modules.

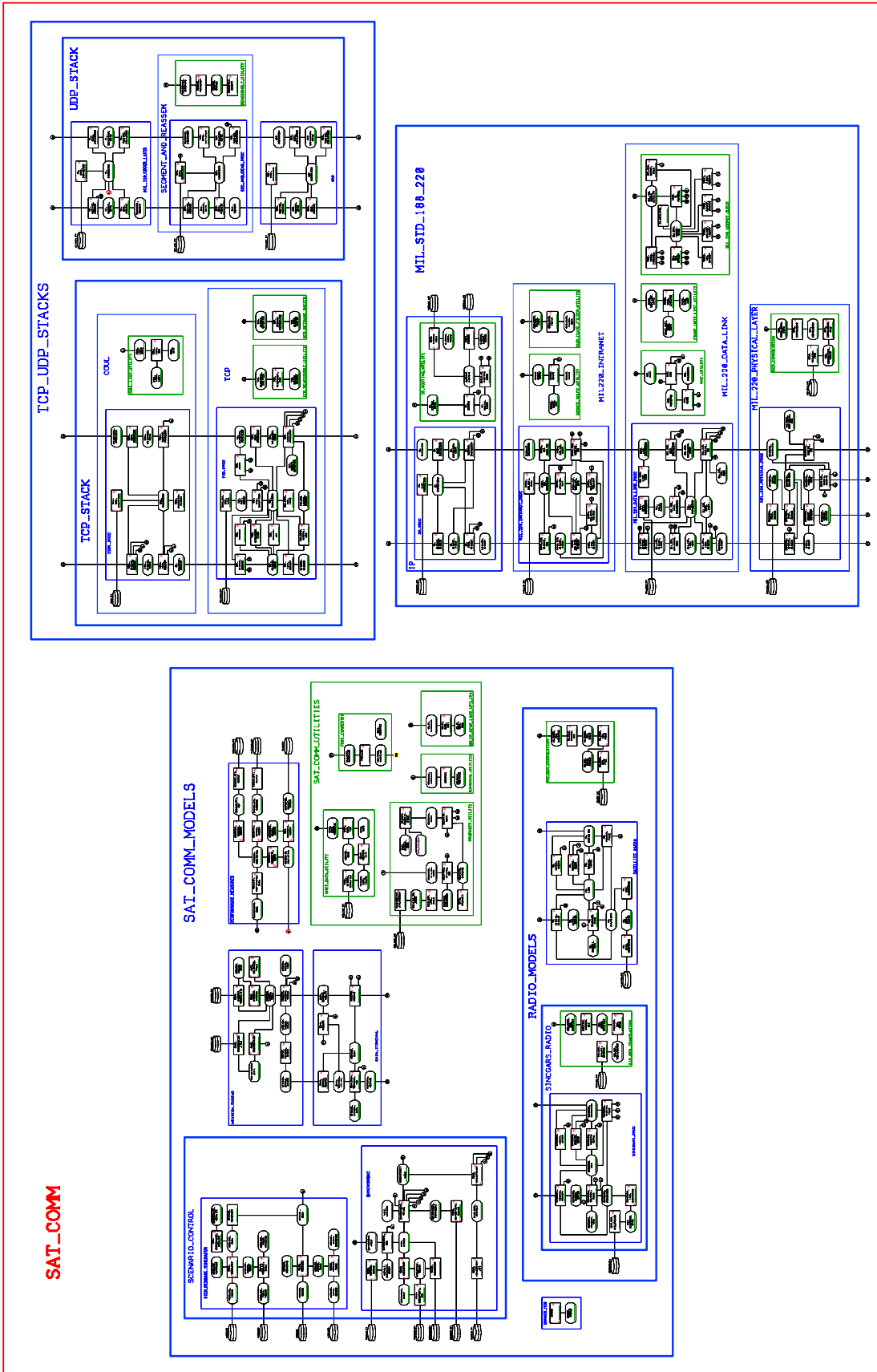


Figure 8-8. Engineering drawing of software

ARCHITECTURAL DESIGN FOR REUSABILITY

In a production environment, many opportunities arise to enhance a module in its original form, or reuse a module in another system. Anyone who has completed a successful project for one client should be aware of the desire to reuse as much of the prior design with new or existing clients. This typically requires changes or additions to the prior design to incorporate new requirements.

We want to highlight those factors that are major contributors to reusability when building a task of the size shown in Figure 8-8 or larger. There is always room for improvement, but this is a good architecture. Telling people that modules should be built to ensure a wide range of functionality, to be easily understood, and to be independent is one thing. Getting a team of developers to carry this out is another. If they have not enjoyed the fruits of having done it before, it will be difficult for them to relate to it! So how does one do this? There are design rules that, when followed, can ensure that module understandability and independence are maximized. Practical experience has shown that certain elements are essential. These are described below.

The competitive environment will eventually dictate that developers who are most productive across the total life cycle will be the survivors. Successful developers know that their products are in the support mode typically for more than 80% of their life cycle. Thus good software architectures must be designed to allow developers to generate new reliable releases quickly.

If most of a software group's time is spent updating an existing system in the support mode, then this is a critical area to analyze to increase productivity. Support includes adding new facilities as well as modifying a growing set of existing facilities. Not only do these facilities grow in number, but they also grow in complexity as more options are added. This is like adding more rooms onto a building. At some point, one must consider the development of a new architecture, else the structure of the old architecture becomes too weak or burdened to support the new facilities. Sticking with an old architecture makes adding new facilities more difficult as a system grows. So how does one develop a good architecture in the first place?

APPROACH TO SOFTWARE ARCHITECTURE

Figure 8-9 below depicts the system life cycle with the detailed architectural design in a blue box. We will define this design process and the results that it must produce. Our long term objective is to define this process in sufficient detail to evolve automation that supports the steps required to complete the design of an architecture.

A Framework For Describing The Architectural Design Process

To accomplish our objective of creating architectures, we must have a framework that affords a sufficiently detailed definition of the process. As before, we will borrow from the mathematics of control theory, using our generalized state-space framework. We will be decomposing the user's functional requirements into a detailed software architecture using hierarchical modules, resources and processes from the VisiSoft version of this framework.

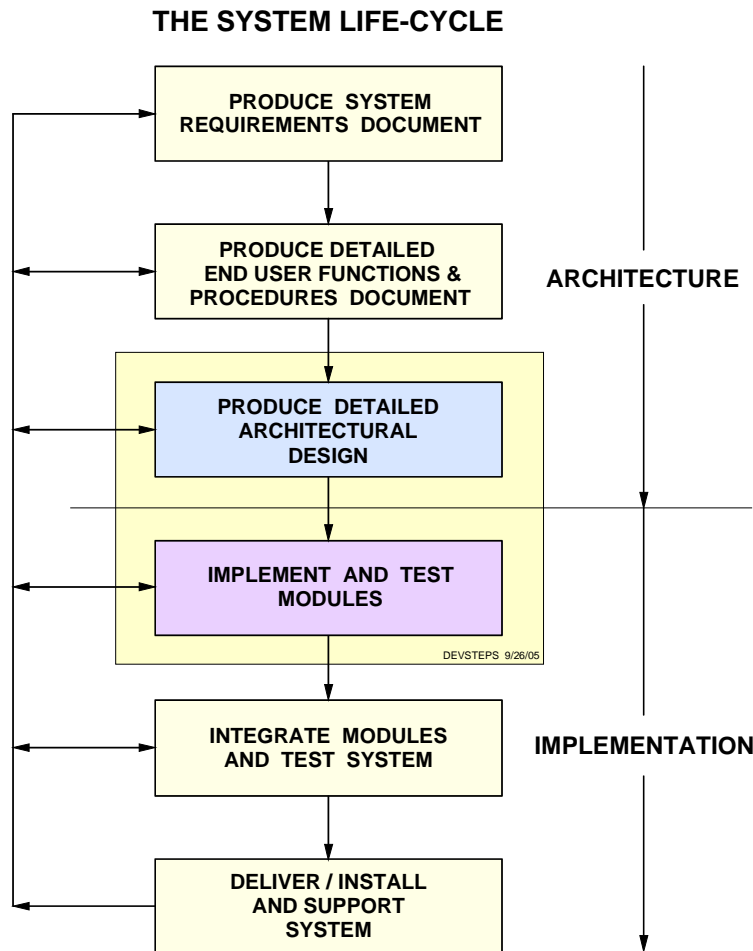


Figure 8-9. Role of detailed architectural design within the system life cycle.

This will be approached as transformations on different substate spaces as well as transformations between substate spaces. Our goal is to produce a hierarchical software architecture, down to the resource and process level, such that there are minimal architectural changes during the coding implementation phase.

The State-Space Framework

The state space framework is a mathematical framework for solving the most general problems in dynamical systems, see [39], [84], or [104]. When providing frameworks for solving such problems, one must ensure completeness and consistency so solutions converge to the expected answer, unambiguously, depending upon their inputs. When developing GSS, the state space framework was used as the basis for ensuring these properties. This was because GSS is a discrete event simulation environment, where flow of control is dependent upon a very large set of event strings that in turn depend upon a huge state space. Although sequences of events are deterministic, they are virtually unpredictable.

The direct mapping of GSS resources into state subvectors and processes into state space transformations made this analogy trivial. The only difference is that GSS state vectors contain non-numeric data in the form of character strings, and the transformations on these vectors permit other than mathematical operations. However, since they are all resolved at the bit level inside a computer, one can show that these operations can all be reduced to the equivalent of mathematical operators on numbers. Therefore, properties of transformations using the state space framework can be applied to those used in GSS.

The state space framework was used to derive fast and efficient approaches to solving circuit design problems in the 1960s, see [47] and [48]. These approaches were implemented in CAD software packages used by engineers for simulation and design optimization of complex electrical networks. When designing this type of software to achieve speed in the multiple simulation optimization process, one is concerned with selection of the best set of state variables for solving the problem, and the grouping of these state variables to minimize the complexity of the transformation.

Selection of the “best” set of state variables equates to choosing the best coordinate system for doing transformations. For example, if one is investigating the dynamics of motion on a sphere, problems are likely solved with much less algebra, and therefore faster, in spherical coordinates. This can be measured using the sum of the products of operations and instruction speeds.

Given the best set of state variables, one may further reduce operation counts by effectively diagonalizing the large matrices that must be inverted to solve the problem. This can be done by interchanging rows and columns of the matrix and corresponding vectors to produce submatrices that are maximally independent. This is known as the optimal ordering problem. This also simplifies the transformations by reducing the weighted operation counts.

Mapping this into the software architecture problem, selection of the best set of state variables is equivalent to choosing the attributes used to represent the states of a system so as to maximize simplicity of the resulting transformation. Optimal ordering is equivalent to grouping these states in a way that further simplifies the transformations. To translate these rules into those for software architectures, a complex transformation may require multiple resources and processes. Selection of the states that represent the software functions, and grouping them into different resources will serve to simplify the processes used to do the transformations. Simplification can be considered from two standpoints: (1) speed of operations, and (2) module understandability and independence. The approach to circuit design achieved both.

Software Architecture Development Steps

We now want to translate these concepts into steps for designing software architectures. Figure 8-10 is an expansion of Figure 8-9. The software architecture development process has been broken into 5 steps. These steps are described below.

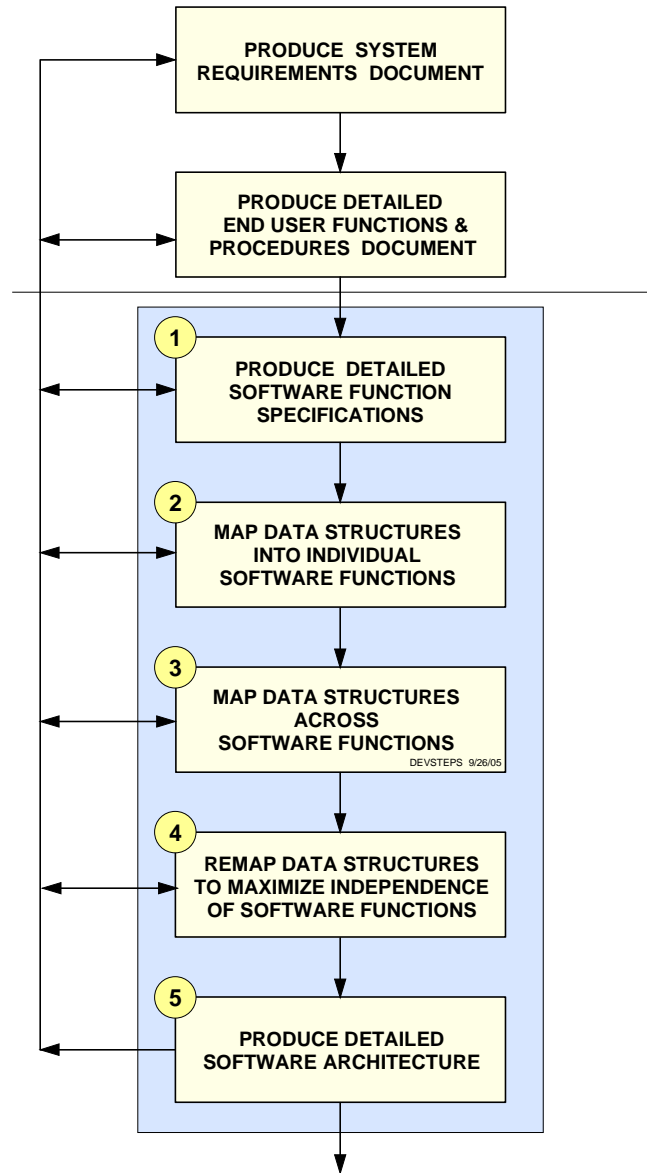


Figure 8-10. Breakout of the detailed architectural design process.

Step 1 - Produce Detailed Software Function Specifications

Determine the generic software functions that are required to perform the end user required functions. Describe these as transformations of state. This implies describing the functional states of the system, and the transformations on those states. In this step we are describing the transformational requirements, and grouping them into an initial set of modules.

Step 2 - Map Data Structures Into Individual Software Functions

Select coordinate systems and states for implementing the software transformations for each function. Map these coordinate systems and states into data structures that support the functional transformations. Assign these data structures to resources and the transformations to processes within modules for each function.

Step 3 - Map Data Structures Across Software Functions

On a module-by-module basis, determine the coordinate transformations required between functions and map the states required to support these coordinate transformations into data structures within resources. Map these transformations into processes. The more complex coordinate transformations may be best put into separate modules.

Step 4 - Re-Map Data Structures To Maximize Software Module Independence

On a hierarchical module basis, review the combination of coordinate systems and states (resources), and the transformations (processes) and determine the best breakout of coordinate systems and states into resources for implementing the transformations. This requires minimizing the resources shared between modules to maximize module independence. This implies shifting data structures from one resource to another, possibly removing some resources and creating new ones. This also requires corresponding changes to the transformations assigned to processes. This implies shifting rules from one process to another, possibly removing some processes and creating new ones.

Step 5 - Produce Detailed Software Architecture

Complete the detailed software architecture by connecting all of the modules. This implies connecting those processes in a module to the resources they share between modules. One may have to iterate between steps 4 and 5 as the architecture becomes more transparent.

The above set of steps represent a great simplification of a complex process. In the next chapter we will provide more insight into approaches to this process. In addition, we will describe utility modules and library modules and how they can be used to further simplify system architectures by removing reusable modules from the main architecture so they can be implemented and tested separately.

ARCHITECTURAL RESTRUCTURING

When users start to use a new system, they immediately see improvements that they would like incorporated into the system. This typically evolves into a life cycle that generates new releases of a product on the order of once or more a year. A good software developer can anticipate many of the potential improvements that customers will want in the future. However, one cannot anticipate unknown desires that turn into requirements. Thus one must be prepared to build architectures that can grow or even be totally revised to meet unknown future requirements.

Chapter 5 described a flexible software life cycle. The arrows in Figures 8-9 and 8-10 illustrate feedback loops that provide for anything to be changed at any time. As soon as one phase is completed and the next phase is launched, one learns about additional features that must be incorporated to improve the system. Flexibility implies that the architecture is designed for ease of change at any level. The ability to do this depends upon the understandability and independence of both the architectural design and the components contained in that architecture.

Mapping Architectural Restructuring Onto The Life-Cycle

Experience has shown that, for complex systems, designers should allow for three restructurings of the architecture during its initial development, and possibly more during support. The reason for these restructurings is because the initial architecture is normally designed without the benefit of detailed knowledge of what the structure must support. Only after going into the additional layers of coding implementation, based upon an initial structure, does one learn about all of the pieces of information to be used and how they must fit together. After one has done this, the first restructuring is likely to be significant. Restructuring will allow the design team to move quickly and easily accommodate the remainder of the design and development effort.

The second restructuring is usually similar to the first, but not as significant. The third is usually represented by a sequence of much smaller improvements to the structure over time. The exception is when levels of module detail have changed significantly since the first restructuring.

Experience has also shown that, having restructured as indicated above, complex modules can be finished, verified, and validated with relative ease. This is because the new structure allows the rest of the information to be added in a much more logical way - increasing understandability and independence. This restructuring does *not* cost time on a project. Experience shows it clearly saves time, particularly when time is most precious - nearing a scheduled release date.

Figure 8-11 shows the effects on a project when restructuring is imposed, compared to the normal project when it is not. When complex models are not restructured, it becomes much more difficult to add additional elements without disturbing large segments of the total module.

DEVELOPING A GOOD DESIGN STRUCTURE CUTS TIME & COST OF PROJECT COMPLETION

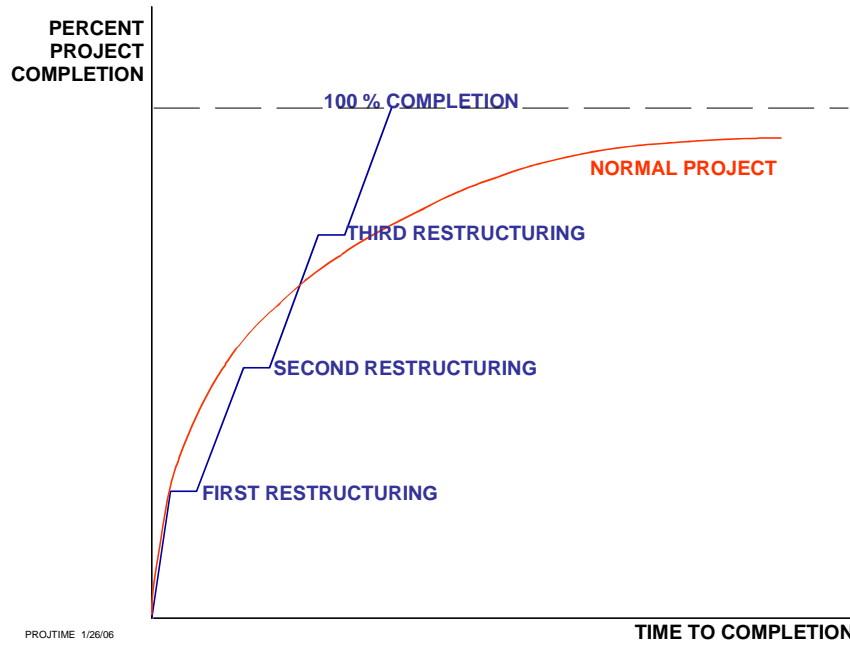


Figure 8-11. The effects of restructuring on project completion times.



Chapter 9. Architectural Design

ARCHITECTURAL DESIGN RULES

Perhaps the greatest benefit of the VisiSoft approach is that the architecture of a complex system can be viewed graphically, and thus studied and improved. Much like looking at the architecture of a house, aircraft, or electronic device, we can view the architecture of a system without being overwhelmed in the detailed design of a particular element. This amounts to hiding the details, e.g., the numerous design parameters, until one wants to see them. In software, this implies not getting into the code. Conversely, if one wants to view the details, they are easily available.

Conversely, the hierarchical architecture hides unwanted details, e.g., numerous design parameters, until one wants to see them. Most importantly, one does not have to read code to figure out architectural details. Finally, if one wants to view the details of the code, it is directly available.

By following architectural rules, software modules can be designed to be independent and easily recognized by others. This cuts the time and cost to build, test, validate, change, expand, and reuse these modules in other tasks or simulations. A sampling of rules is provided below. These rules are generally easy to follow and verify, simply by reviewing the engineering drawings.

Some Basic Rules

Ensuring independence and understandability at different levels of an architecture is an important part of ensuring reusability. This requires understanding what is meant by *independence at different levels*. The following are the principles behind this concept.

- Independence and understandability of an *elementary* module allows it to be copied and reused, with modifications, in another part of the drawing or in another *drawing*.
- Independence of a *higher level* module provides for ease of reuse in different drawings or in different directories for different *projects*.

The following are basic design rules that, when followed, can ensure that module understandability and independence are maximized. For example, we want to ensure the following.

- Resources shared between two processes contain only those attributes that are *shared by* those processes.
- Attributes used by only one process are placed into a resource *dedicated to* that process.

These rules maximize understandability and independence at the elementary module level. They are time-saving practices that are easy to follow. Even the following simple rules help.

- Do not reuse working variables by more than one process.
- Use separate working variables for different functions, *each using meaningful names*, even within a single process.

These rules add significant improvements to the understandability and independence of elementary modules. Additional rules are provided in the following sections.

Limiting The Number Of Elements Within A Module

An elementary module may have only one or two elements, e.g., a resource and process, although this does not occur often. However, when a process or module starts to expand in size as more detail is added, one must consider breaking it into separate processes or submodules for clarity of understanding. Three to five elements allow a module to be easily understood. Unless the structure is very symmetrical, and therefore still understandable, one should limit the number of elements of a module to ten.

Limiting Interfaces To Two Interconnect Lines

As shown in Figure 9-1, modules can be connected via many interfaces. For example, module M2 is connected to modules M1, M3, and M5 via three two-line interfaces. M3 is connected to module M2, M6, and M4 via three two-line interfaces. However, at every interface there are no more than two lines interconnecting any modules. Often, one line will connect to a resource for input data, and the other line to a resource for output data.

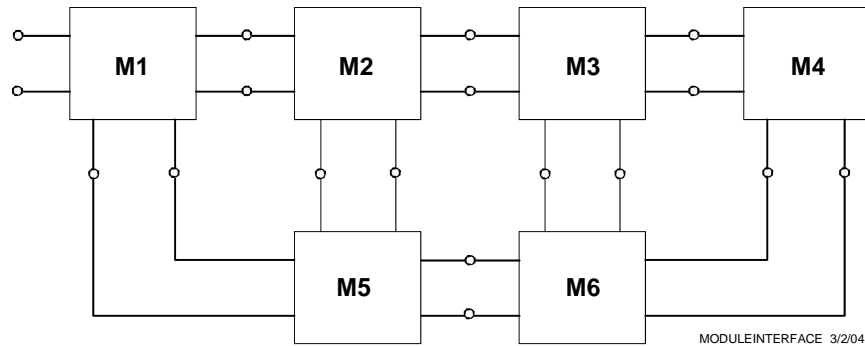


Figure 9-1 Module Interconnection scheme.

Limiting the interconnections between modules to no more than two lines at any interface is the most important way to achieve module independence. It allows one to easily make a copy of the module and connect it to other modules in other drawings.

BASIC ARCHITECTURES

Simple Architecture 1 - Elementary Design

Figure 9-2 illustrates the most basic architecture of a task. In this case, the task, C_S_1, contains a single module, M_1. The module, M_1, contains one resource R_1 and one process P_1. To indicate the use of interactive DISPLAY and ACCEPT statements, a computer terminal is attached to the process. The computer terminal icon is for documentation purposes only.

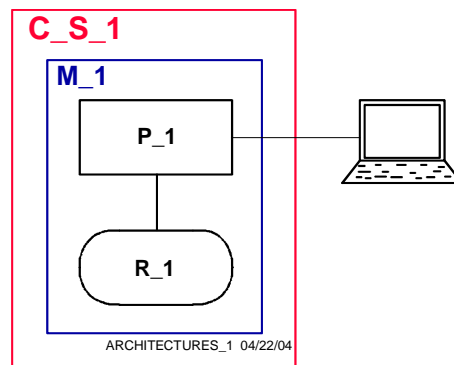


Figure 9-2. The most basic architecture of a task.

Simple Architecture 2 - Two Modules

Figure 9-3 illustrates a task with a simple two module architecture. From the nature of the architecture, M_1 is the main control module, and M_2 is a subordinate module. The implication is that P_11 calls P_21.

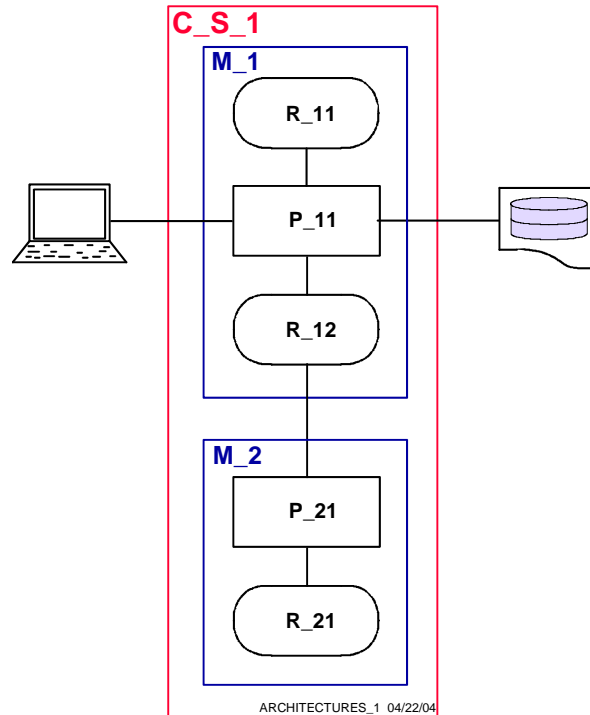


Figure 9-3. A basic two module architecture of a simulation or task.

Note that both processes have a dedicated resource (P_11 has R_11 and P_21 has R_21). This is based upon the principle that a process generally has a need for attributes that are not shared with any other process. The rule to be followed is that, *unless attributes (data) must be shared for functional purposes, do not share them*. Use a dedicated resource to house those attributes, even for a single attribute. When memory was expensive, programmers learned to share work variables. This is prone to misunderstandings and bugs. Today, memory is cheap. Time is very expensive.

When one process calls or schedules another process, these processes generally share data. If resource (R_12) contains only the data shared between these processes, this maximizes independence and understandability. If a module boundary is crossed, the resource should be placed inside the controlling module.

When an experienced architect looks at this drawing, the above principles are expected to be followed.

Use Of Terminals

In conventional engineering drawings, terminals are used to connect lines between drawings, or to simplify a single drawing. Rather than have a long line connecting elements across a drawing, one that may interfere with many other lines, a label is created at each element to denote the connecting line. Figure 9-4 illustrates this convention.

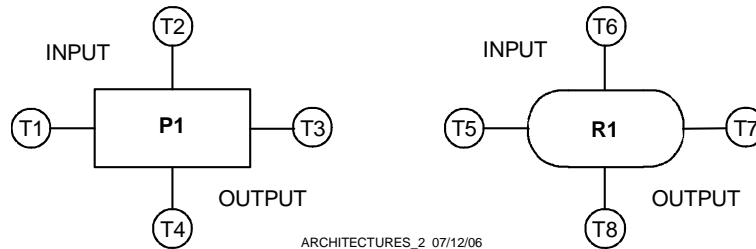


Figure 9-4. Use of labeled terminals.

Looking at Figure 9-5, we note the following conventions. Information and control inputs generally come in on the left and top (T1, T2, T5, and T6) and outputs generally go out on the right and bottom (T3, T4, T7 and T8) for both processes and resources. Here, if P1 is called, the resources it shares with the caller are considered inputs to P1, and those resources would be attached at T1 or T2. If P1 calls another process, then resources it shares with the called process are considered outputs of P1, and those resources would generally be attached at T3 and T4.

Looking at R1, T5 and T6 are typically tied to processes that call others tied to T7 and T8, where the calling process is providing inputs to R1 to share with the called processes. This does not prohibit data from being shared in both directions, where one need only consider the direction of control.

Architecture 3

Figure 9-5 illustrates the architecture of a higher level module M_A that may be used by another module at an even higher level. Module M_A contains four submodules, M_1, M_2, M_3, and M_4. Rather than packing all of these resources and processes into one module, it makes sense to break up the functions so that they can be isolated and therefore treated independently. This is the job of the architect. Specifically, modules M_2, M_3, and M_4 can be dealt with on a reasonably independent basis from the rest of the module.

The connector to A indicates that a resource in the higher level module contains data that is used and modified by M_A.

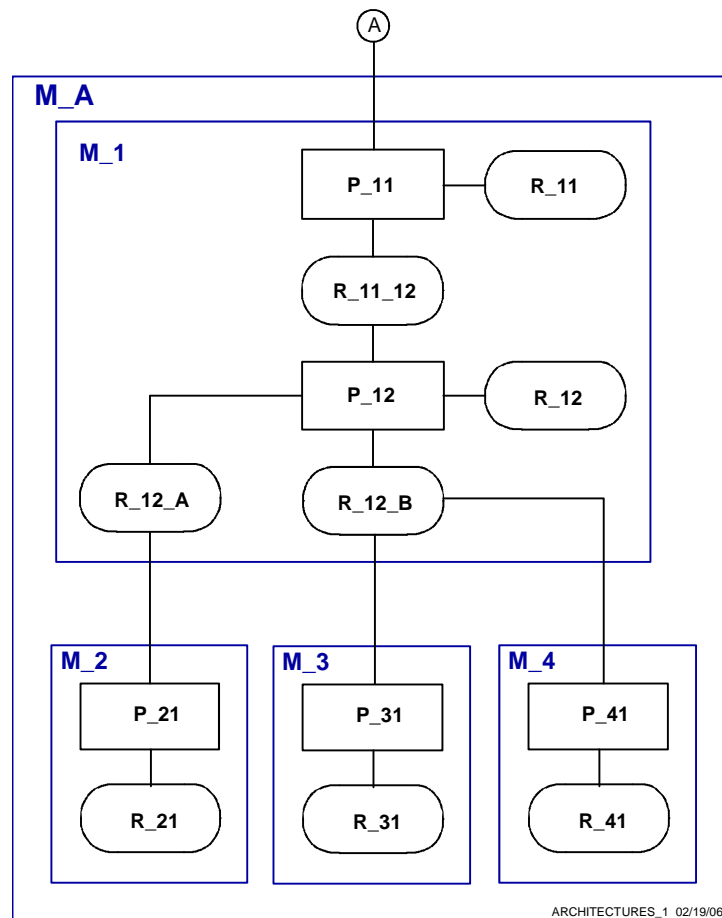


Figure 9-5. Architecture of a higher level module.

ADDING SHARED ALIAS RESOURCES TO GAIN INDEPENDENCE

Sometimes it appears difficult to limit the interconnections between modules to just two interconnect lines. This problem typically occurs when a resource in one module is to be shared with more than two processes in another module. Figure 9-6 provides an example of this case, where MODULE_1 calls four processes in MODULE_2, while sharing a single resource with those processes.

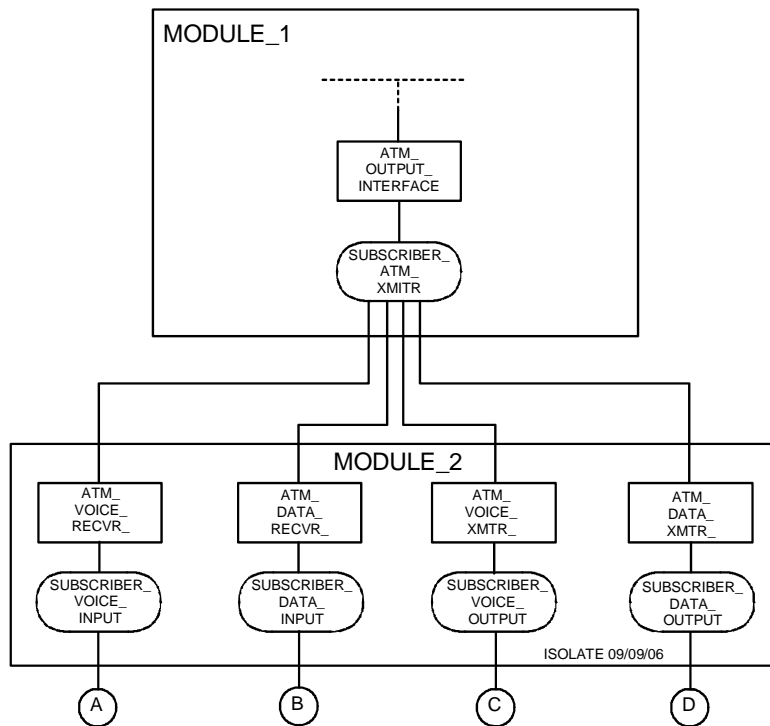


Figure 9-6. Example of *not* limiting the interconnections between modules.

This problem can be solved by adding a SHARED ALIAS resource in MODULE_2, as shown in Figure 9-8. The SHARED ALIAS resource (denoted by the red outline) is ATM_SUBSCRIBER_INTERFACE and is connected to the four processes in MODULE_2. When the process in MODULE_1 calls any of the processes in MODULE_2, it uses the memory of resource SUBSCRIBER_ATM_INTERFACE, now a SHARED AS resource, as shown in Figure 9-7.

The SHARED ALIAS resource does not use separate memory, but points to the memory in the SHARED AS resource SUBSCRIBER_ATM_INTERFACE. The “pointer” is implicit in the architecture and not of concern in the language. The data structure within the SHARED ALIAS resource acts as a template that is overlaid on top of the memory defined by the SHARED AS resource.

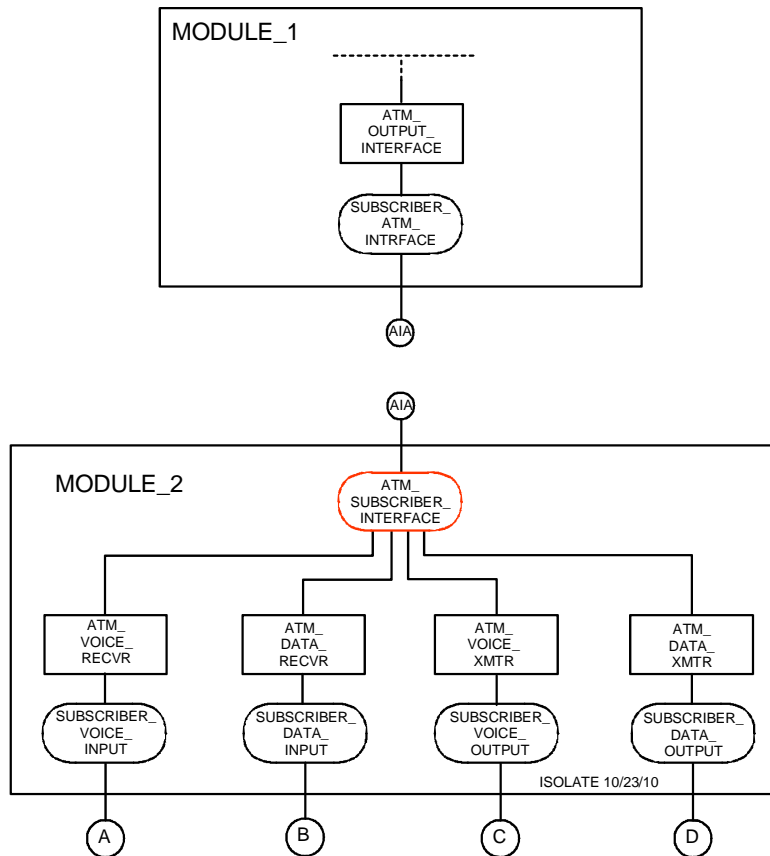


Figure 9-7. Example of how a SHARED ALIAS resource reduces the interconnections.

At first, this architecture may appear to carry unwarranted overhead. However, it is the facility needed to ensure independence between modules, and requires no additional memory. It occurs in many design situations. It is particularly useful when designing Utility and Library Modules as defined below.

SHARING RESOURCES WITH INDEPENDENT UTILITIES

Frequently, as indicated above, more than one module must perform the same or very similar functions. Rather than repeat the code, the desired function may be put into a single module. More generally, when two or more processes in different modules call the same process sharing similar data structures, it is best to create an independent utility. This is done by putting the shared data into a SHARED AS resource associated with each calling process, and creating a Utility module with a SHARED ALIAS resource.

This requirement is illustrated in Figure 9-8. MODULE_1 and MODULE_2 each want UTILITY_6 to perform a function using their own resources, SHARED_AS_M1 and SHARED_AS_M1 respectively. Instead of sharing both resources directly, UTILITY_6 shares them using SHARED_ALIAS_U6.

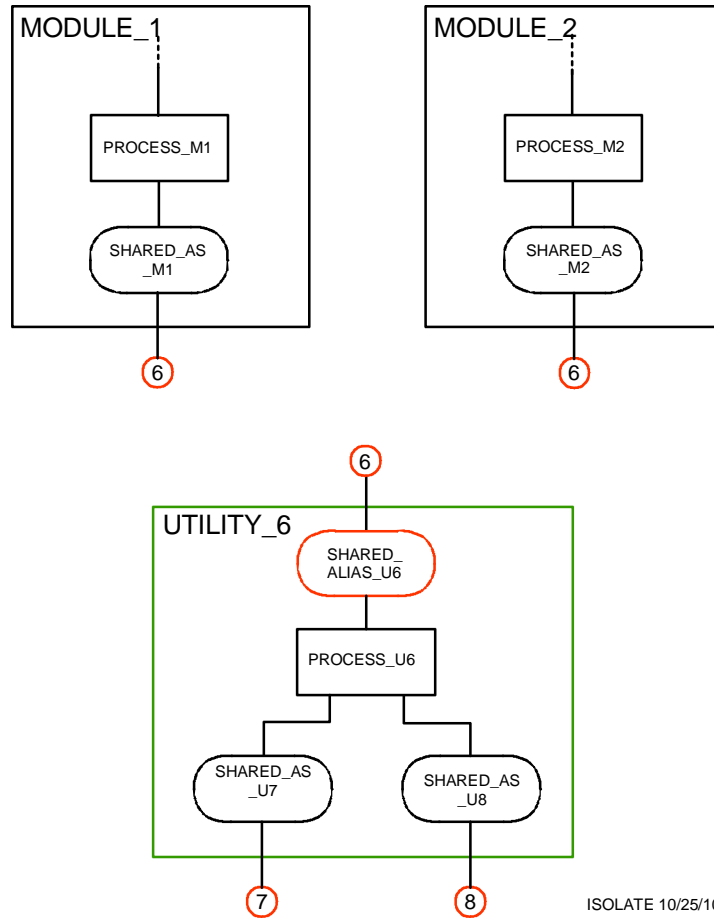


Figure 9-8. Example of using aliased resources to create independent utilities.

Thus when MODULE_1 calls upon UTILITY_6 to perform the function, SHARED_AS_M1 is used. When MODULE_2 calls upon UTILITY_6, SHARED_AS_M2 is used. If UTILITY_6 is *memory-less*, no logic is required to distinguish between the two callers and their needs. Also, no direct connect lines may be drawn to a Utility module. Utilities must be connected to other modules via connectors.

Figure 9-8 illustrates an additional facility in VSE and GSS. This is the ability to chain utility or library modules. Resources SHARED_AS_U7 and SHARED_AS_U8 can act as SHARED AS resources when attached to other Utility or Library modules.

BUILDING PROTECTED DATABASE UTILITIES

Figure 9-9 illustrates an architecture for a *Utility* (green border) that takes in an input file and provides access to a database that may be available to multiple modules. This architecture provides for reading FILE_I. Since the file is an input file, it is shown on the left side of the module. Process PI controls access to the database, initializing the database that is stored in RIDB.

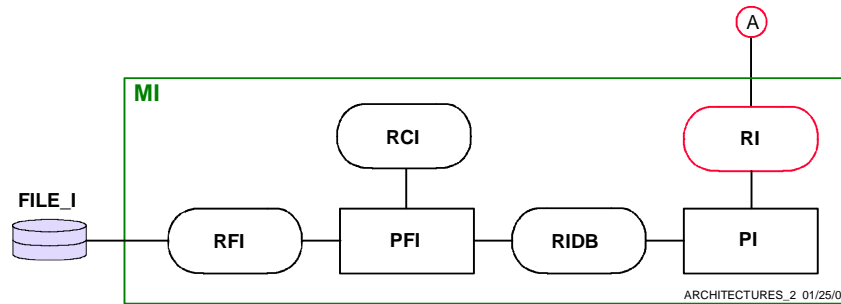


Figure 9-9. Architecture for an input file handler utility.

Process PI is called from above to retrieve data from the database. The first time it is called, PI calls PFI to read the file and load the database. PFI reads records from the file, loading the database into RIDB. When all of the records are read, PFI sets the END_OF_FILE status and any other return codes and returns control to PI. PI then performs retrievals requested from above. If the request coming into PI from above is to update the database, PI performs this update to RIDB. When the task is completed, PI may be called from above to write the updated database to the FILE_I (if desired) and close it.

RFI contains data structures representing the records read from the file. RCI is used to hold control attributes used by PFI. RI is a template that holds control and database information stored in the module above.

Figure 9-10 illustrates an architecture for a database handler utility that can store an output file. This architecture provides for writing FILE_O when PO is called from above to do so. Since the file is an output file, it is shown on the right side of the module.

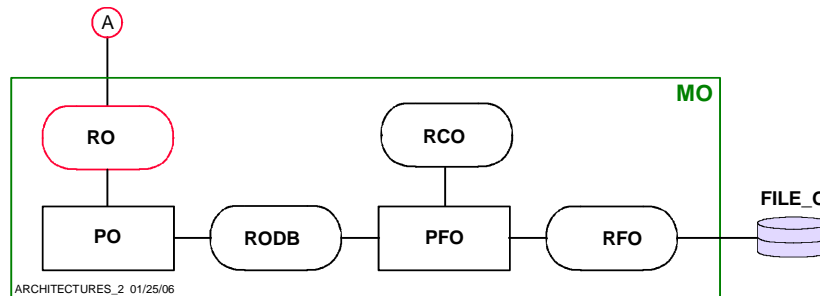


Figure 9-10. Architecture for an output file handler utility.

BUILDING INPUT-OUTPUT FILE HANDLERS

Figure 9-11 illustrates an architecture for reading and writing Standard File Interface (SFI) files. The SFI standard was created by GSS users in the 1980s. SFI files are text files whose data records follow similar formats. They also contain predefined header records to describe the format of the fields in the data record. Software is automatically generated to read and write SFI files based upon the header information. This allows the file to be connected directly to a process that reads or writes the file automatically when called. Fields in the data record that are specified by the header records must have a match with those in resources that are connected to the calling process. These fields are automatically updated when reading an SFI file. Users can create SFI formats that are easily read from - or read into - spread sheets, statistical packages, etc., that use simple delimiters between fields.

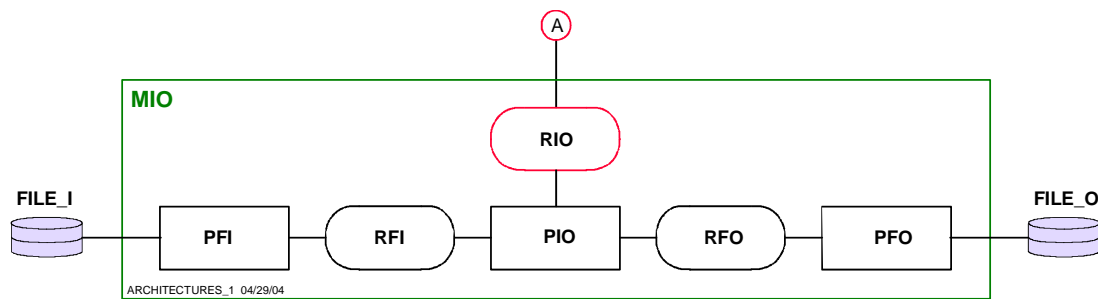


Figure 9-11. Architecture for a file handler utility.

The architecture in Figure 9-11 provides for reading FILE_I into a SHARED AS resource in the calling process using RIO as the SHARED ALIAS template and then writing that database to FILE_O. Process PIO controls the module, initializing the database that's stored in the SHARED AS resource that's connected to RIO and the calling process. PIO can create a copy of the file when required by calling the process PFO to write the database to FILE_O. The calling module then performs database retrievals and updates directly to RIO.

In general, process PIO is called from above to retrieve the database from FILE_I or store it in FILE_O. The first time it is called, PIO calls PFI to read the entire FILE_I. PIO calls PFI once for each record and loads the information from RFI into the database in RIO. PFI reads records off the file into RFI. When all of the records are read, PFI sets the END_OF_FILE status and any other return codes required for PIO to send back to the calling module. The calling module then performs retrievals directly from its own SHARED AS resource.

When the task is completed, PIO is called from above to write the updated database to file FILE_O and close it. To write FILE_O, PIO moves data records from RIO to RFO and calls PFO to write the file for each record.

STANDARD LIST UTILITY ARCHITECTURES

Figure 9-12 illustrates examples of standard architectures for list utilities that are quite common in database work. These particular architectures are used to manage different types of lists or databases that support different pieces of complex software.

The CAD system described here contains a GENERAL library of modules that can be used to perform various standard software functions. These are described in the General Library document, reference [79]. Examples of popular library modules are those for managing static and dynamic (linked) lists, e.g., those shown in Figure 9-12. Management of linked lists can be somewhat complex for large applications where they may be required in many different parts of an application. In this case, one can build utility modules to use the list management library modules. Once one has a utility built, one can easily modify the module, process, and resource names, and put the new utility module into a different piece of software. One can then proceed to tailor the code where necessary to achieve the new functionality. Most of the code changes are required to match record layouts in the lists (databases). The rules are usually unchanged, except for some of the data names. The logic is generally bug-free on the first implementation.

Probably the most important concept to be derived from this is that there are standard architectures that are recognizable directly from their drawings. Once one gets to work with these different architectures, it is clear how they can be permuted to do different functions very easily. This is because the common functionality has a corresponding common picture (architecture or sub-architecture).

Without visualization of these architectures, this would be intractable. Imagine that your eyes were only limited to reading text, and could not recognize pictures. Or if the pictures were gross abstractions of the functionality (e.g. a box without any processes or resources inside) so that the picture does not tell very much about what is inside. That is a major difference between this CAD approach and other software approaches, except for those based upon flow-charts. But as the logic gets complex, flow charts expand in size, typically covering many sheets of paper, becoming hard to follow. A good language can easily reveal complex algorithms on one or two pages that may take five or ten pages of flow charts.

USER_LIST_UTILITIES

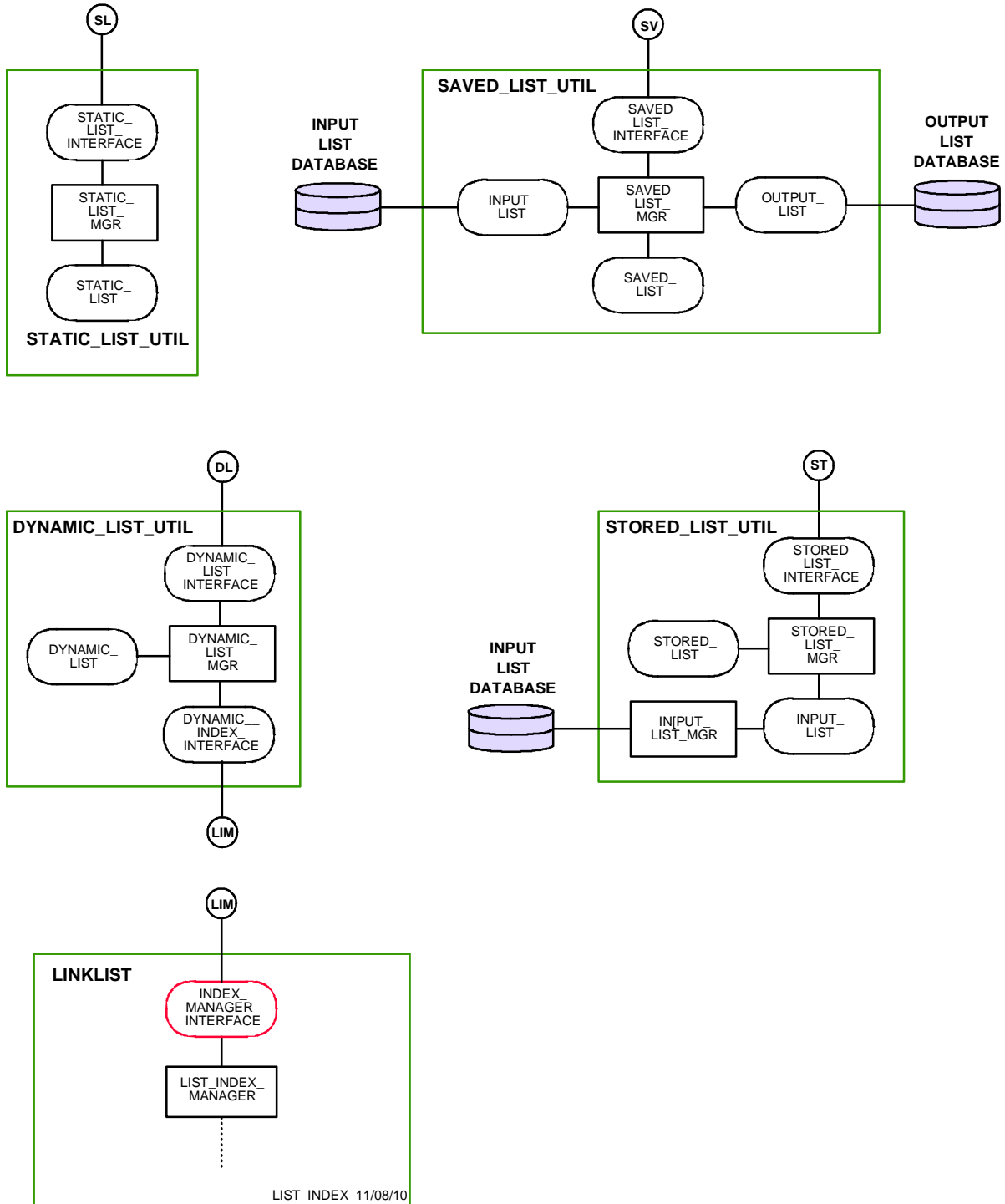


Figure 9-12. A few examples of list management utility architectures common in VSE.

BUILDING PANEL ARCHITECTURES

Using this CAD system, interactive panels are built using the Panel Library Manager (PLM). One must first create a *panel resource*. Then one can select this resource and bring up the PLM drawing board to create a new panel or modify an existing panel, e.g. RESOURCE_PANEL shown in Figure 9-13. When the panel is completed and saved, the panel resource is populated automatically with data structures produced by the panel drawing board. This panel resource is then available to processes that are connected to it architecturally. At run time, these processes can put information into, and get information from panel fields within the data structure.

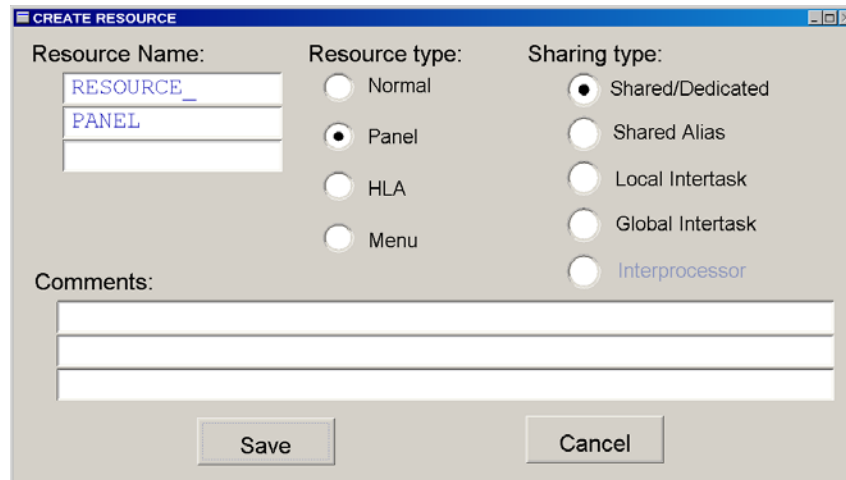


Figure 9-13. Creating a Panel Resource Type.

BUILDING PANEL HANDLERS

One of the many features of the CAD system is the ability to create panels such as that shown in Figure 9-13 using a graphical environment. At run-time, users may enter data, choose selections and review outputs using a large number of built-in features. The panel shown in Figure 9-13 happens to be used to create a resource that provides the interface between all of the panel fields and the task that is being created to use a (different) panel. These resources are special panel resources that appear in the architecture with their names in red. Our interest here is the architecture for multiple panels, not the panels themselves.

Figure 9-14 illustrates an example of an architecture for handling multiple panels. This architecture is much better because the process PANEL_CONTROL becomes very complex if the panels themselves have any degree of complexity.

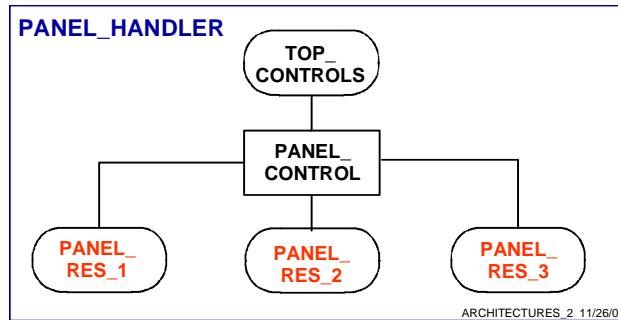


Figure 9-14. Poor architecture for handling multiple panels.

When taking in input data from multiple panels, managing which panel is open and on top should be separated from accepting and editing data from the individual panels. Figure 9-15 provides an example of an improved architecture for accomplishing this.

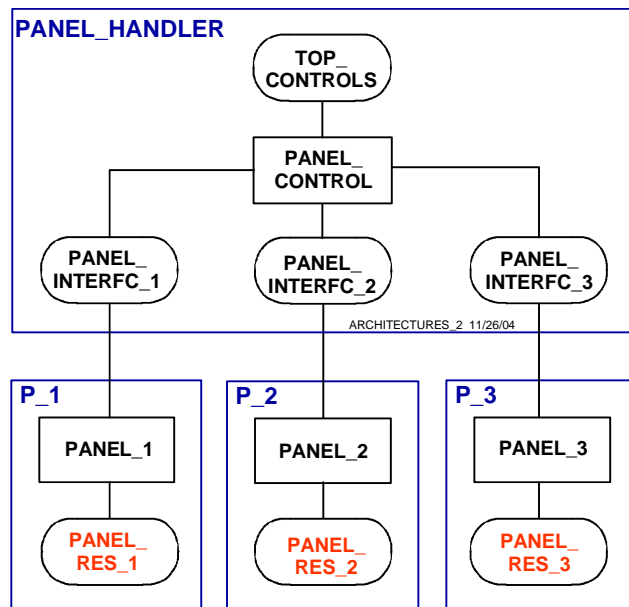


Figure 9-15. Improved architecture for handling multiple panels.

Figure 9-16 provides the best example of an architecture for handling more complex panels. It is recommended that the architecture used for P_3 be considered for all panels. The resources used in this architecture each serve a different purpose.

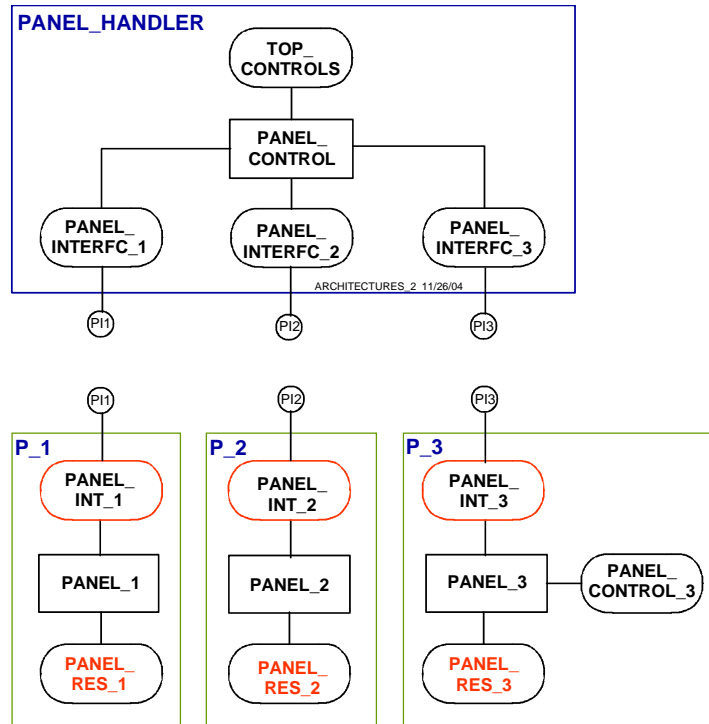


Figure 9-16. Best architectures for complex panels.

PANEL_INT_3 serves as the interface to the calling process and allows the panel to be tested using a test driver independent of the system or simulation in which it resides. PANEL_CONTROL_3 provides for internal work or control attributes used to control the logic fielding the panel inputs and responses. PANEL_RES_3 is the panel resource itself.

ARCHITECTURES FOR TESTING LIBRARY MODULES

Architectures for testing library modules can be simplified by following the procedures offered below when building the modules. Figure 9-17 illustrates a module POINT_AND_VECTOR_TESTS that is contained in the GENERAL library. This library module contains three utility modules that are used for determining the relative positions of points, lines, and vectors in space. Because these utilities contain complex heuristic algorithms, they require tests using substantial databases to ensure all possible situations are covered. When testing complex library modules in a production environment, one may run many tests before certifying a module for production release.

A test driver for the utilities in this library module is shown in Figure 9-18. Because library modules are prepared as object files (in this case it is the GENERAL.a library), having copies of the utility modules available inside the test driver task eliminates having to go back and forth from the test driver to the library to make the changes. As utilities, they can be put into more than one drawing in a the library directory. Changing a utility on any of the drawings changes it everywhere. So when the utilities are finally corrected in the test driver in Figure 9-18, one goes into the library drawing and prepares the library module for a final test.

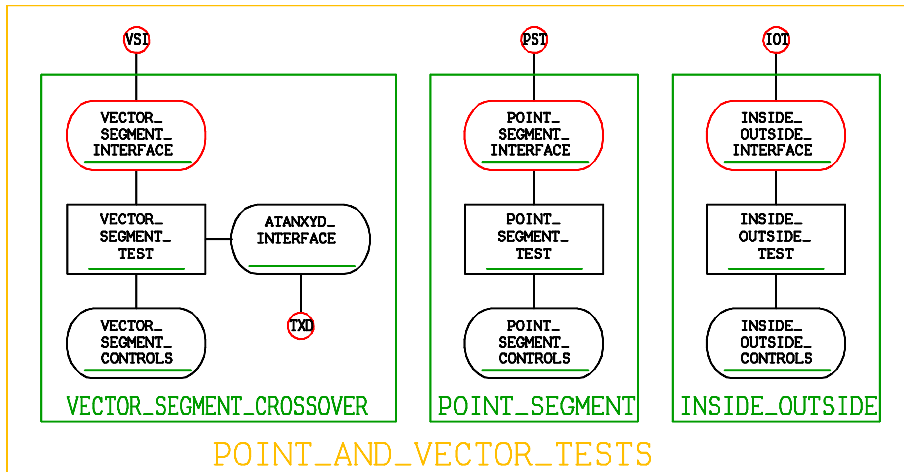


Figure 9-17. Library modules under test.

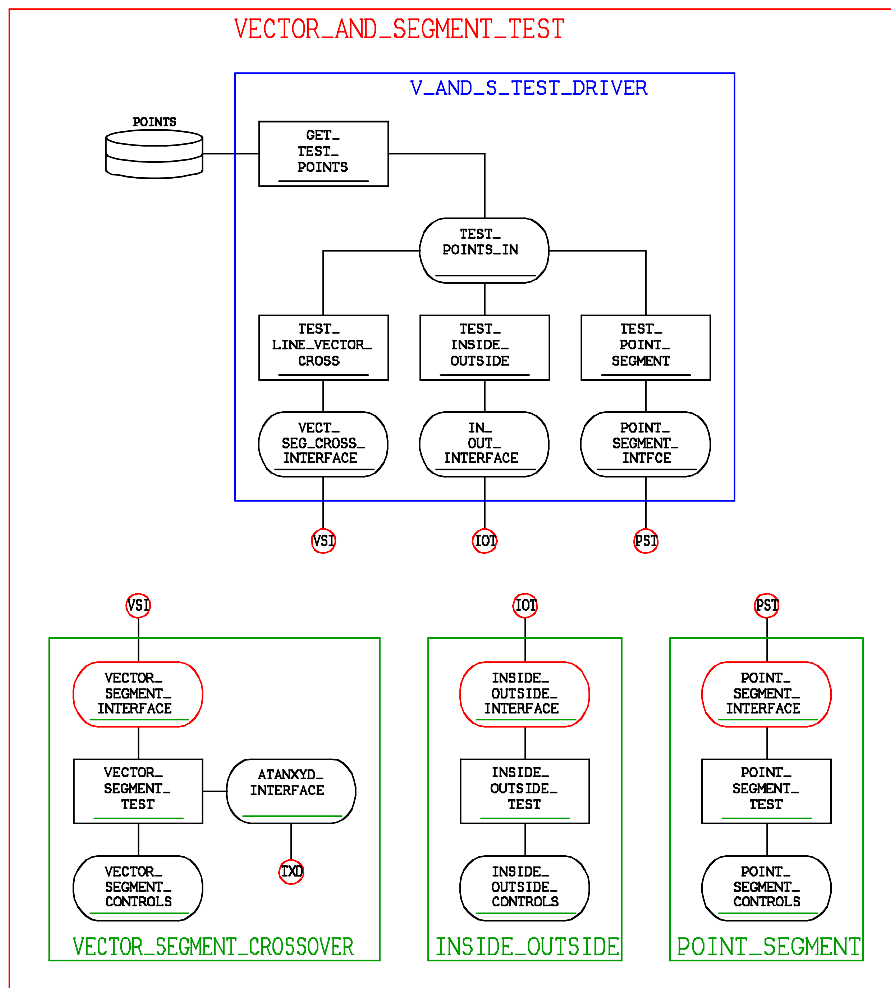


Figure 9-18. Test driver for library modules.

Having generated a new corrected version of the library, an example of a call statement inside the test driver is shown below.

```
CALL INSIDE_OUTSIDE_TEST IN POINT_AND_VECTOR_TESTS IN GENERAL
    USING IN_OUT_INTERFACE
```

To invoke the test procedure described above, one merely has to put comments into this call statement to call the utility modules inside the test driver as shown below.

```
CALL INSIDE_OUTSIDE_TEST *** IN POINT_AND_VECTOR_TESTS IN GENERAL
    *** USING IN_OUT_INTERFACE
```

After testing is complete, and the library module is prepared, then the comments are removed and the actual library module is called as a final test.

PUTTING RESOURCES ON THE TRANSMITTER SIDE

When designing systems where information is moved between modules, the resource containing the data to be transmitted across the boundary (e.g., RS1) is best placed inside the sender module with the process that receives it (e.g., PR2) inside the receiver module. This approach is illustrated in Figure 9-19 below.

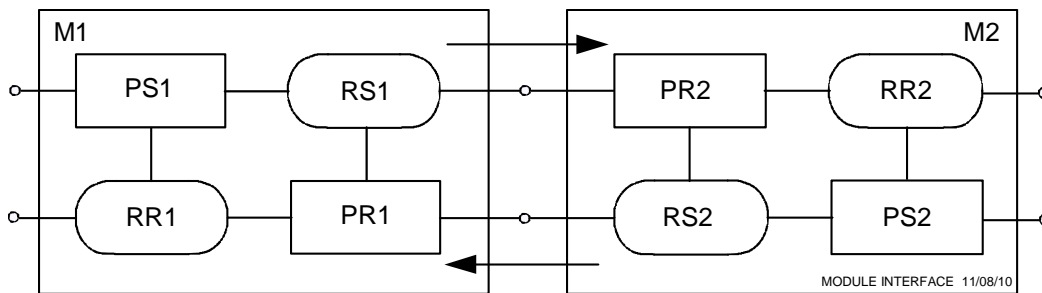


Figure 9-19 Module Interconnection scheme.

The reasons for this approach are many-fold. First, when designing communications systems, information is transmitted between two boxes as shown. For asynchronous transmission, the transmitter puts the information to be transmitted into a buffer in the transmitter, and sends a control signal to the receiver to sense the information and copy it into the receiver. This is accomplished by having the transmitter schedule the receiver module to copy the data after it is put into the transmitter resource shared with the receiver.

For synchronous communications, the transmitter schedules itself to put the information into the buffer at specified clock times, and the receiver schedules itself to sample the transmitter's buffer in between those clock times.

ARCHITECTURAL SPACING

There are many disciplines in which engineering drawings are used to represent a design. House architectures, machine drawings, and chip layouts are examples of drawings that are to scale. We are only concerned with drawings that are not to scale, e.g., electronic circuit designs and logical designs. In both cases, engineers typically follow standards for production drawings. This is because large systems cannot be represented with a single drawing. It is not unusual to have ten or more drawings to cover the components of a production system. Thus, the drawings must match to be read easily, and engineers are known for their adherence to these standards. This has nothing to do with style. It has to do with clear representations of the design.

Good architects joke about the different approaches to spacing between modules and elements within a module. In fact they have been given names, e.g., Texas - wide open spaces, and China - crunched up. Is one way better than another? Are there benefits of spreading things out versus crunching them up? The answer is definitely yes. Good spacing practices assist understandability and ease of architectural change.

When working with very large complex drawings, one can distinguish between good and bad practices. For example, a technician creating the drawing on a graphics screen may want to crunch everything up to fit as much on the screen as possible. This minimizes the number of drawings. But engineers reading these drawings may have difficulties discerning where one module ends and another begins, especially if they are not familiar with the design.

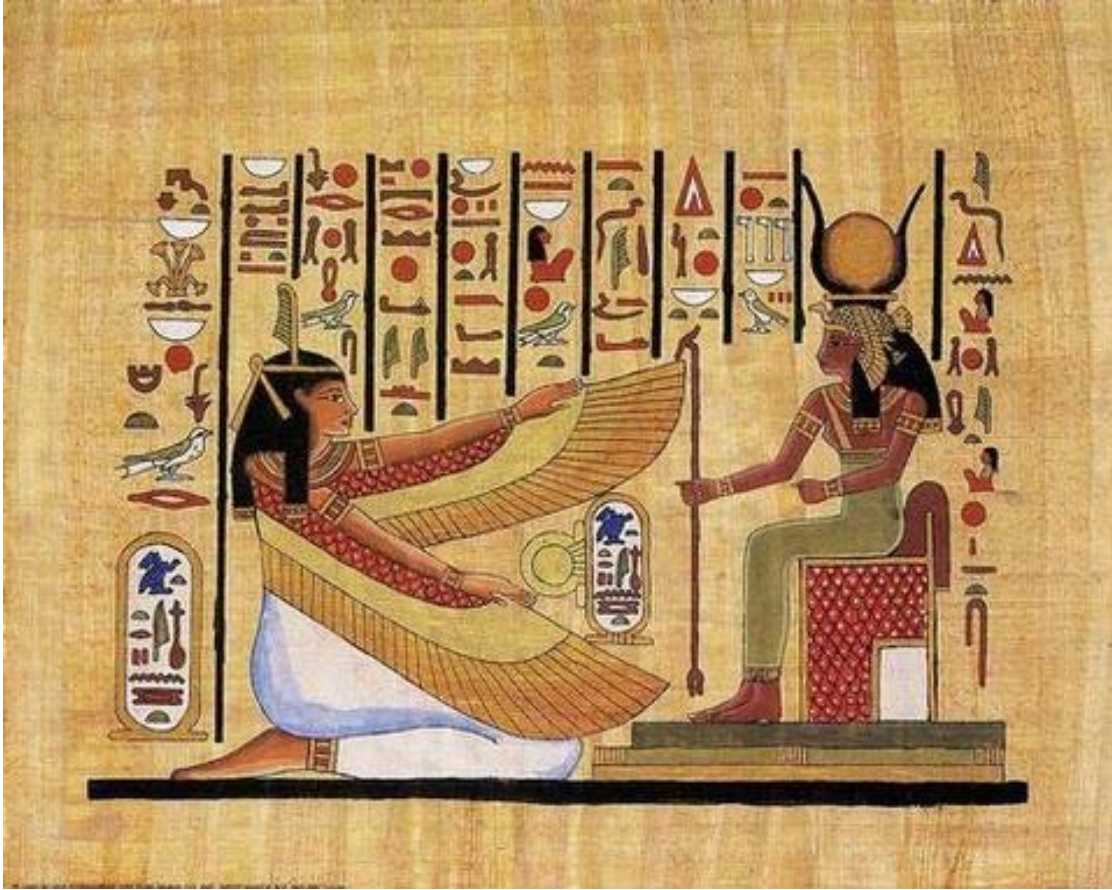
In fact there have been studies relating productivity to pixel counts or size of the drawing one can fit on a screen. The more pixels one has, the more one can see on the screen with the same degree of clarity. Productivity has been shown to increase with pixel count.

In addition, having visited aircraft manufacturing plants, one sees huge drawings wrapped around the walls, with moving step ladders to view the drawings. Large complex designs can never be put onto a single screen. So crunching things up is not the answer. What is important is the ability to easily pan and zoom on a large drawing, and to easily break a large design into multiple drawings. A large design will naturally have independent parts, else it is a poor design.

Changing An Architecture

Even more important is the time expended when one wants to make an architectural change. If the drawing is crunched, it becomes a large task to make some room, just to be able to move module boundaries - to create some additional space inside for one or more new elements.

To save time changing a module whose architecture is crunched, one starts looking for ways to not change the architecture. This problem is typically solved by adding code to existing resources and processes. These elements start to take on additional functions that make them difficult to understand. When additional functions are best put in separate resources and processes, or even separate modules, one is faced with changing the architecture. Having space to start with is extremely helpful, and will likely make the difference between creating an improved versus degraded architecture.



Chapter 10. Language Concepts

This chapter and the following three chapters address the language issues in VisiSoft. In this chapter we look at concepts behind the languages, with special attention to two key issues:

- Support for the architecture
- Understandability of the code

We must also consider:

- Speed of execution.
- Independence of sections of code (within a process)

Speed affects productivity in the run time environment. It is especially important in more advanced applications, for example large scale simulations, graphics, and particularly complex algorithms. The independence criterion allows us to read one section of code without regard to another section of code.

Language Support For The Architecture

Software has always been built by programmers. These are the people who generally sit at computer terminals and write the programs (code) that make computers do what they do. But this is like thinking of house-builders - people who get together and build a house. In fact, long ago, this is the way houses were actually built. There were no architects, masons, carpenters, plumbers, electricians, sheet-rockers, etc.

Why do we have these separate professions and fields of skill today? It is much more productive. The old adage “Jack of all trades, master of none” applies. In fact, it also applies to software. As we have seen in the prior chapters, architecture plays a vital role in structuring and supporting large scale systems. The skill set required for designing architectures is far greater than that for writing code. An architect designing a house must specify how the different components must fit together. This requires a knowledge of the tools and materials that can be used by those who will implement the architecture. Likewise, the software architect must know the approaches that programmers can take to write the code, selecting those as appropriate for the design. An example is where to use a utility module or a library module.

To support the architectural approach used in VisiSoft requires three separate languages. Data cannot be declared in the process language, instructions cannot be declared in the resource language, and neither can be declared in the control specification language. This is another major paradigm shift. Having three separate languages has allowed the language designers to focus on the role of each. The result is that the languages are a major factor contributing to higher productivity in general, as well as the architectural properties of the system.

Understandability Of The Code

It is our belief that one should not require a programming background or a degree in Computer Science to be able to read and understand complex algorithm descriptions. We know this is possible from our experience with VisiSoft. One may not know the syntax required to write a program, but one only needs the appropriate subject area expertise to read and understand what the algorithms are supposed to accomplish.

Conventional programming languages encourage minimum keystrokes and terse (economy of) expressions. One may have to read a line of code multiple times to figure out what it means. This is the opposite of reading prose, where good authors take pain to ensure the reader can quickly capture the ideas. If one cannot read fast, one may lose the “train of thought.” This does not help an author trying to sell books.

Furthermore, when we read prose, upper and lower case are used to provide more information. This information comes from redundancy, a remarkable trait of the English language, a language that has survived and is used more than any other. It has become the language of international trade. One of the reasons is its understandability. Much of this is due to its redundancy. Grammatically, articles, such as “the”, “a”, etc. help to ensure that the reader understands what the writer meant. They could be dropped and one could still derive the meaning. So why use them? - To make the sentence more understandable.

Although understandability was considered very important to programming back in the 1960s, today it represents a vast change, one that can open up the software environment to a much wider degree of professionals. The VisiSoft departure - to emphasize readability and understandability - has great benefits for the person reviewing the algorithms. VisiSoft has put understandability as far more important than the brevity of source code.

Our approach to various language issues is based upon arguments from the earliest days of programming onward, having learned and used many different programming languages. The resulting conclusions are often significantly different from C based languages, including C++ and Java, the only languages taught in most universities today. Moreover, it is our contention that one should not require a course in a language to be able to read and understand an algorithm, as is often required with languages like C++ and JAVA. This allows subject area experts to validate a model, or verify that what is built is what is needed. Most important, the VisiSoft approach has been evolving in a production environment since 1982, with many changes and refinements, specifically to accomplish the speed and productivity goals.

Speed Of Execution

Speed affects productivity in the run time environment. It is especially important in more advanced applications, for example large-scale simulations, graphics, and particularly complex algorithms. So what effect does language have on speed of execution? It can be significant. A typical response by an experienced programmer seeing VisiSoft code for the first time is that the language must be very inefficient - implying that it must run slow. After clocking large algorithms, one is surprised at the speed improvements. Two contributing factors are (1) the ability to easily describe large hierarchical data structures, and (2) the ability to move large strings of bytes into a data structure that contains a large number of different field layouts - without concern for word-boundary alignment. These facilities support handling large records from files or complex message structures very rapidly.

Independence

The conscious separation of data from instructions in VisiSoft has provided significant benefits. First and foremost, it allows one to capitalize on the concept of independence. The independence criterion allows us to read one section of code without regard to another section of code.

Properties of independence have guided software development in the past. Early on, engineers who were designing large CAD systems in FORTRAN tried to achieve module independence in software for the same reasons they did in hardware - to minimize the effects that design changes in one part of the system would have on the other parts. This was accomplished by minimizing the number of modules (subroutines) that shared the same labeled common blocks (data). The approach is illustrated in Figure 10-1. The independence properties could be determined visually by looking at the matrix of modules that used labeled common. *The more sparse the matrix, the greater the independence.*

This was not a new concept for engineers. They had learned it in Linear System Theory. The more sparse the matrix that represents the system (of equations), the more independent the variables, and *the more simple the transformations*.

Also on these projects, unlabeled common was never used. Its use was considered a bad practice since it tended to be global, and destroyed independence. Good practices were those that minimized the creation of bugs, especially in the support phase when new features and functions were being added by new programmers.

Less experienced programmers put everything into unlabeled common. One change affected all modules. They also used argument lists. These proved to cause problems in the support phase, since all calling routines had to specify every argument in the proper order. By using labeled common, one only need refer to the name of the common block.

MODULES \ LABELLED COMMON	CONTROLS	RANMX	MODEL C	MODEL D	TITLE	DESCRIB	PRINTX	INPUTX	IOCTRL	PRCTRL	DIRECT	SAMPLE	STRSRS	FSNAME
MAIN	X	X	X		X	X		X	X	X	X			X
CONTRL	X	X	X						X					
GUIDE	X	X	X						X					
SIMUL8	X		X							X		X		
RANDOM		X	X						X					
DSCRIB						X			X					
PRINT	X		X						X					
MODEL8			X	X	X	X	X		X	X			X	
PRINT			X				X		X	X		X	X	
DEBUG	X	X	X						X					
GETSMM								X	X	X	X	X		
PUTSAM									X	X	X	X		
GINPUT								X	X					
COVAR	X	X	X	X				X				X		X

LABELLED_COMMON 1/31/06

Figure 10-1. A module independence matrix for FORTRAN.

Some of these engineers ended up writing business programs in COBOL, and carried over the same separation and independence principles using very similar matrices. However, it was much easier because of the inherent separation of data structure statements from procedure statements in a program or subroutine in COBOL. In both COBOL and later versions of FORTRAN, one could use INCLUDE statements in the subroutines by naming the INCLUDE files containing the detailed data structures.

THE THREE *VisiSoft* LANGUAGES

As mentioned earlier, the VisiSoft environment has three separate languages:

- Separation of data from instructions is accomplished by describing all data in a *resource language* and describing all instructions in a *process language*.
- Independence from the platform and operating system is accomplished by defining system aspects using a *control specification language*.

Resources, processes, and control specifications are *prepared* (translated) separately.

Benefits Of Separate Resource, Process, and Control Languages

Although this may appear unusual, the benefits of having separate languages quickly become apparent. Conventional programming languages generally combine the specification of data with the procedural language statements. As a result, data definition takes a back seat to the procedure logic that tends to overwhelm thoughts of data organization. Programmers rarely get to appreciate the fact that *logical procedures can be greatly simplified by using well-organized data structures*.

Having a separate language and visual container for defining data elevates the importance of building structures that are well thought out in advance, not something that is invented “on the fly” - as needed. Structuring becomes second nature when one has a language that makes it easy to define and easy to use. This is the antithesis of other languages. One must use VisiSoft to understand these benefits, because they are otherwise nonexistent, and therefore hard to fathom.

Processes also benefit from having the data elements they use defined in a visibly connected resource. It is not unusual to have three or four windows open to see the resources as well as the process being created or changed. More importantly, good data structures simplify the procedural statements that use them, and are a major factor in understandability of complex conditional situations.

The control specification language eliminates the need for OS level scripts to assign files and run a program. Everything runs under the VisiSoft environment or as a separate executable (that can invoke other executables) on any operating system. This provides platform and OS level independence.

The Benefits Of Separating Data From Instructions

In VisiSoft, modules can be readily designed to be independent. By limiting access to specified data structures, a module will achieve true independence. Data structures are defined in *resources*. Instructions are grouped into sets of rules defined as *processes*. The interconnection of processes and resources is done while creating the engineering drawing of a module’s architecture, where lines connecting processes to resources determine what processes have access to which resources. Modules can be connected to each other by connecting a process in one module to a resource in another. *Independence of modules can be inspected, visually, simply by looking at the number of lines connecting them*.

This also allows one to decompose a database into separate data structures. Resources and processes can be grouped into elementary modules. Elementary modules can be grouped into hierarchical modules. This is illustrated in the prior chapters. Resources, processes and control specifications can be edited directly from the engineering drawing.

As indicated above, the separation of data from instructions allows two separate translators: one for data structures, and one for rule structures. This separation provides a natural language for describing the rules. Language design is not aimed at economy of expression (minimum keystrokes to enter lines of code), or ease of writing the “compiler”. Instead, language design is aimed squarely at ease of understanding by other than the original author - the property of *understandability*.

To make the rules readable requires that the data structures and corresponding typing be defined to support ease of understanding the rules. This is most important when trying to understand conditional statements. For example, consider the following statements:

```
IF A(1) == ", "
```

Versus,

```
IF FIRST_CHARACTER IS A COMMA
```

If one does not know the C language, one may be hard pressed to understand the first statement, while the second statement is obvious to anyone who knows English.

As will be apparent from this and the next three chapters, the three languages are a significant departure from the terse languages designed to make compilers small and simple to write. Having three separate translators eases the burden of individual translator design, but that is not the driving force. The driving force is understandability.

SOME REFLECTIONS ON CONVENTIONAL LANGUAGES

The programming world went through a period of enlightenment in the late 1960s and 1970s. During that period, many papers were written on the benefits of top down design, structured programming, one in-one out control structures, and data organized into hierarchical structures by name - specifying the type after each name. These were important revelations. Unfortunately, there were no general-purpose languages that supported the concepts professed to improve software development.

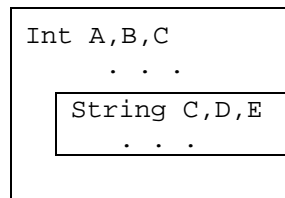
COBOL had already implemented many of the desired features with its paragraph orientation and its record structures facility. But the implementation of these features left much to be desired. Scientists could not effectively use COBOL and academics shunned it since it did not support scientific applications. COBOL also took programming to a vocational level, cutting salaries by a factor of two, causing a dislocation in programmer employment. PL1 implemented some of the features of COBOL, but was really a scientific language.

An even greater impact was created with the huge expenditures by AT&T to get into the computer business. Billions of dollars were spent promoting UNIX, and C went along for the ride. With Bell Laboratories dominating the IEEE and other software journals, the end result has been the huge promotion of C. Although COBOL and FORTRAN are still used in the inner circles of commercial and scientific programming, everything else is written in C and its derivatives - C++ and Java. Most academic environments require no other languages.

The authors have listened to many complaints from older programmers, with experience in many languages, regarding Computer Science graduates having no knowledge of languages other than C and its derivatives. These graduates lack the knowledge of languages that were much better designed and much more productive. There are sufficient articles now being published on the lack of a real engineering discipline in software. We believe there is a renaissance on the immediate horizon. We hope we can contribute to it with this book.

The Quagmire Of Scope In Conventional Languages

In conventional languages, data and instructions are grouped together into blocks, functions, or classes, variables can be declared within nested constructs. With two blocks we can have a scenario as follows:



Here it is easy to see that the variable C in the inner block is a string, not an integer. This brings up the twin issues of visibility and scope. In simple terms, when we see a name in a program, we want to find the declaration that defines its properties. The region of text over which a declaration applies is its “scope”.

Scope is a pervasive issue in programming languages. It is interesting intellectually, and convincing in certain ways. Our first problem with scope comes with the complexity of the scope rules themselves. The above scenario greatly oversimplifies the practical effects of scope rules in the context of all language features.

A parameter of a function has a scope that is local to the function, thus hiding outer occurrences of the same name. Functions may also introduce local variables. Both of these cases are similar to the simple block structure above.

C++ and Java both allow inheritance. In this way, a “base” class can have a “derived” class. The derived class retains properties (functions and variables) from the base class. Classes can have public, protected, or private elements, and this itself adds a different kind of scope rule.

A derived class begins with an access specifier, which controls the visibility of the base class variables within the derived class. The derived class can have a public, protected, or private access specifier. The individual elements within the derived class can be either public, protected, or private. This gives rise to different combinations.

For example, if the access specifier is public, a public member of the base class is visible in the derived class, but a private member of the base class is not. There is definitely a logic here, although keeping track of these rules in practice can be difficult, and reach to a high mental overload for the program reader.

C++ also allows one to name a sequence of declarations; this sequence is called a "namespace" and can be referenced elsewhere in the program. Continuing on deeper into scope issues would bring us to an almost endless tour-de-force in complexity. This makes for great academic exercises, but is clearly at odds with productivity.

The Scope Productivity Problem

The productivity problem is this. One is revising a piece of code, and encounters a name. What does the name refer to?

In theory, this is easy. We look up the hierarchy of structures and namespaces for the declaration of the name. Note that the name may be externally defined, either through import or a visible namespace. Sounds easy, but not so.

Practically, the problem runs more like this. The name itself will have some semantic content. Let us say the name is "SP_DeblockWord". We need to find out some detail about this name to understand the meaning of the program text. Where is the information we need, i.e., where is the declaration?

Our "program" is, say, 120,000 lines, our module is, say, 2,000 lines. Questions:

- Is it a method or function, or a variable?
- Is it an integer? Or is it a pointer to an integer? Or an array or structure?
- Does sp stand for saved pointer
- Is Deblock to be read as a verb, "to de-block" or as an adjective, short for "deblocking"?

To continue interpreting the code we are revising, we need this information. It is not obvious where the declarations is, and the region of text where it might be is quite large. The practical result? The declaration of SP_DeblockWord is lost somewhere in our program.

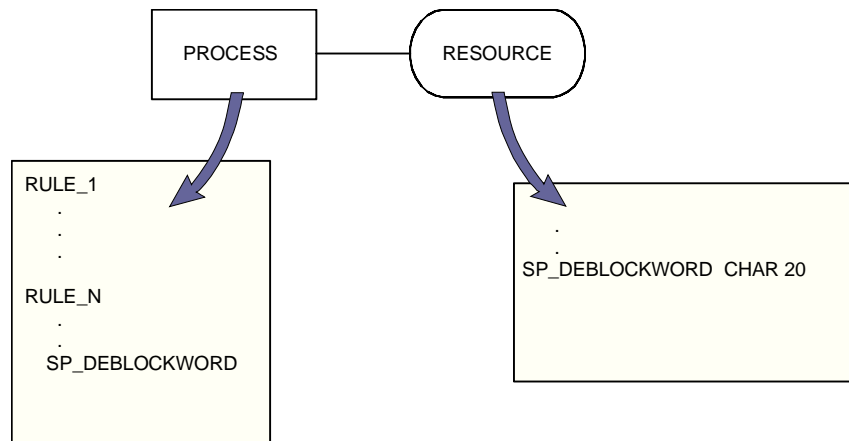
There may be a hundred names or more. We may need to look up many of them in order to proceed at all. A huge impediment to going forward.

THE VisiSoft SOLUTION TO SCOPE AND VISIBILITY

Using VisiSoft, the solution to the scope issue is based upon the requirement that:

A name used in a process must be defined in a resource directly connected to the process.

That's it! The resource contains the declaration of the name. Pictorially, we have



The readability and understandability of this one simple rule affords the clarity of detail needed to support easy maintenance, implying improved productivity. There are two significant factors in achieving this. The first is that

- The connection of a process to a resource is directly visible in the architecture.

The second is that

- All possible places to look for a name are directly visible from the architecture itself.

There is no need to worry about scope, local vs global, or different rules for visibility. In fact, *data is never global in VisiSoft*. Processes can only share data as specified at the architecture level. Independence of a design is directly determined by the way processes share resources. In *VisiSoft*, we see who shares what data on the engineering drawings. It is not hidden within the language level. If we want to hide complex modules so as to not be distracted, we can cover them up. But if we want to drill down to understand them in detail - we can do so, easily!

When programmers first look at VisiSoft, they typically have the misperception that data is always global. This also causes them to conclude that data cannot be passed by "pointer." Actually, the converse is true. Data that is shared between processes at the architectural level is always accessed via pointer. Passing by value is extremely inefficient. If a designer in VisiSoft desires to make a copy of data to protect the original, he simply moves a copy of it to another structure.

A NATURAL NOTATION FOR PROGRAMMING

To appreciate the elegance of the notational approach in VisiSoft, we first briefly review conventional approaches to notation.

Current Notation

Since the advent of ALGOL in the academic world and C in the UNIX world, specialized notation has been a part of almost every programming language, especially those featuring lexical scope and object-oriented features. This terse approach to notation carries over to almost all programming languages. Thus C++, Java, C#, Perl, etc. have a terse style. For example:

C++

```
do {
    position = to_lower( page ).find( href, position );
    if ( position != string::npos ) {
        int link_beg = page.find( "\", position );
        int link_end = page.find( "\", link_beg + 1 );
        string link;
        int index = 0;
        link.resize( link_end - link_beg - 1 );
        for ( int i = link_beg + 1; i < link_end; i++ )
            link[index++] = page[i];
        . . .
    }
```

Java

```
public void actionPerformed(ActionEvent e) {
    if ( e.getSource() == BtnShowResult ) {
        // checking for erroneous input, begin
        String InputString = TxtFldPhNo.getText();
        char InputArray[] = InputString.toCharArray();

        if ( InputArray.length != PHNO_MAX_SIZE ) {
            JOptionPane.showMessageDialog(null,
                "Length not equal to "
                + PHNO_MAX_SIZE, "Invalid length",
                JOptionPane.PLAIN_MESSAGE);
        }
        return;
        . . .
    }
```

PHP

```
echo "<table border='1' cellpadding='2' cellspacing='0'
width='400'
align='center' id='MyApplication1'><tr>";
```

For a programmer reading pages and pages of C++ or JAVA code, one's mind must always be parsing the text in order to get to the deeper semantic issues that underlie the particular application.

Anyone who has read the history of C, e.g., [56], [63], knows that the two major factors affecting the design were (1) the compiler had to be easy to write, with a terse (easy to parse) syntax being justified by economy of expression, and (2) the compiler had to fit into the very small memory of a PDP-11 computer. This was because the project was not really supported by the management at Bell Labs at that time. What the authors accomplished in the face of daunting constraints on their computer environment is worthy of being acclaimed as a great feat. However, it is our view that the resulting notations are also daunting to use, minimizing characters used (not necessarily time to type), and fit only for those desiring artificial job security. This style of programming hardly fits what would be considered intuitive to the average engineer.

The New Notation

VisiSoft adheres to this basic concept:

- The notation reflects familiar natural language constructs and minimizes specialized notations.

As a result, the mental overhead of parsing the VisiSoft syntax is greatly reduced. This is because the language is context oriented. This puts the burden on the translator, which takes effectively nine passes to generate code. But large pieces of software are translated with the blink of an eye. The labor has been put into the translator in order to make it easy for the software developer.

Moreover, in VisiSoft, many issues are resolved so that the awkward behavior of traditional languages is no longer an issue. The result is a significant improvement in understandability of the resulting code.

We start with the most simple VisiSoft assignment statements:

```
INCREMENT DAY_COUNT BY 7          (or DAY_COUNT = DAY_COUNT + 7)
DECREMENT TOTAL_LOSS BY GROUND_LOSS
ADD 100.3 TO TOTAL_LOSS(LINK_POINTER)
SET AIRCRAFT_STATE TO ON_THE_GROUND
PERCENT_BUSY = (TOTAL_BUSY_CALLS * 100)/TOTAL_CALLS
```

Although not earthshaking, the style of arithmetic reflects ordinary usage, i.e., anyone can read and understand it - without knowing a programming language. In fact, VisiSoft arithmetic is virtually identical to FORTRAN, including embedded complex arithmetic. This puts programmers concerned with their professional status and corresponding job security ill at ease.

For control structures, an issue discussed in a separate chapter, we also get a clean simplicity of style:

```
IF CALL_TYPE IS LOCAL
    INCREMENT TOTAL_LOCAL_CALLS.

IF OUTGOING_LINE IS NOT BUSY
    EXECUTE CONNECT_CALL.

EXECUTE NEXT_CALL 5 TIMES

EXECUTE READ_MESSAGE UNTIL LEAD_CHARACTER IS A DELIMITER

READ EXTERNAL_FILE
    AT_END EXECUTE SYNTAX_CHECK

IF UPDATED_ADDRESS_FILE EXISTS
AND UPDATED_ADDRESS_FILE IS NOT EMPTY
    ASSIGN UPDATED_ADDRESS_FILE TO OUTPUT_FILE_RESOURCE.
```

The general direction of the syntax will become more evident in the examples that follow. The three most important factors in developing the VisiSoft syntax were:

understandability, understandability, and understandability !

USE OF UPPER CASE - LOWER CASE

One can notice that the examples of VisiSoft given here use entirely uppercase letters. This is deliberate and, in fact, required. Why? What are the motivations for using lower as well as upper case? In prose, one starts the next sentence immediately after the prior, as opposed to putting each in a separate paragraph as we do with code? Therefore, to flag where a new sentence starts, one uses capital letters. But why in a programming language?

Solving The Great Library Mystery - Using Upper And Lower Case

A major motivator for lower and upper case in programming stems from large library systems. Working with X-Windows, one quickly understands why the library routines all have long names, with upper case - lower case, that are almost impossible to type correctly the first time. It is to help ensure that one links to the correct library routine. X-Windows libraries contain hundreds of C functions, and each of these must have a unique name. This is because, if one writes a program in C or one of its derivatives, one must be concerned that functions contained in two or more linked libraries may have the same name. When this happens, it is not discovered until run time - when odd things start to happen - the result of linking to the wrong routine. Current linking conventions, being the mystery that they are, link to routines with the same name, but in different libraries, in what appears to be a *random order*.

VisiSoft provides a library facility that guarantees uniqueness at link time. This is accomplished by requiring that all library functions within a library module have unique names, and that all library modules within a library have unique names. The user must name the library and the module as well as the function. This hierarchical uniqueness capability, built into the VisiSoft library facilities, is a major feature, eliminating the need for long unique library function names.

Prose vs. Programs

It is well known, and can be proved experimentally, that reading text is easier when the font uses proportional spacing and both upper and lowercase are used. That is, conventional text is most readable when we use both upper and lowercase in a proportional font.

Fixed width fonts (e.g. Courier, Courier New) are normally used for computer programs. Our contention is that programs are not like text, but rather like mathematics or tables. Fixed width fonts ensure alignment of code, making it easier to read. Using a proportional font makes good layout difficult, and therefore difficult to read.

Consider the proportional code

```
char c1;
c1 = GetChoice()_____ ;
switch (c1)
{ case 'a', case 'A': ProcessOptionA(); break;
  case 'i', case 'I': ProcessOptionI(); break;
  case 'w', case 'W': ProcessOptionW(); break;
  default: cout << "Not a valid choice\n";
}
```

With a fixed width font we have:

```
char c1;
c1 = GetChoice();
switch (c1) {
    case 'a', case 'A': ProcessOptionA(); break;
    case 'i', case 'I': ProcessOptionI(); break;
    case 'w', case 'W': ProcessOptionW(); break;
    default:          cout << "Not a valid choice\n";
}
```

Conventional programming languages make things even more difficult when we start using mixed upper and lowercases. It is not uncommon to see requirements, for example, that constants be written with all uppercase, class and type names start with an initial capital letter, and variable names start with a non-capital letter.

These kinds of conventions can be extended to different special cases, where the use of capitalization becomes significant as far as interpreting what a name actually means, for example

```
public class ConsumeAlert extends Thread {
    private JTextArea output;
    private HoldAlertSynchronized cHold;
    private JTextField fireL[], intruderL[], commonL;
    public boolean Terminate = false;
    public ConsumeAlert( HoldAlertSynchronized h, JTextArea o,
        JTextField FireL[], JTextField IntruderL[],
        JTextField CommonL) {
```

This puts a strain on both the readability and modifiability of the resulting module. Somehow, important properties are supposed to be conveyed by the use of capitalization. Such conventions are difficult for both the learner and the reader. To compound matters, compound names make use of capitalization to separate words. This also sets up another series of conventions that may or may not be followed.

The Upper Case Approach

This leads us to the VisiSoft convention, where the names are all uppercase. Compound names are separated by an underscore. The resulting programs are eminently easy to read. This convention is both easy to learn, understandable, and clearly more productive.

Programs are not stories that are only read by the reader. They are modified by the next programmer who has to add new functions and features to a product in the support phase. To do this, one must use existing attribute names as well as add new ones. This implies that one must ensure that old names are not reused improperly. When looking at long names with upper and lower case, it is difficult to remember what is upper case and what is lower case. Anyone who has worked with X-Windows understands this as the library management problem - addressed above.

Programs are thus more like tables. Tables use monospace fonts, and capitalization is not an issue. Programs also contain mechanical algorithms - a cross between an algebraic statement and a logical statement. They are supposed to convey a clear statement, not subject to interpretation. But they contain more than mathematical statements. They have complex IF ... THEN ... ELSE ... statements imbedded, that may represent very complex logic, logic that would take huge logical expressions if represented in a basic language for logic.



Chapter 11 Data Structures

GENERAL HIERARCHICAL STRUCTURES

Architectural decomposition of a complex system requires an effective breakout of the states and transformations comprising the system. Just as we can deal more economically with organizations that have a hierarchical structure, we can deal better with applications whose software is organized in a hierarchical form. Complex system states are generally defined as hierarchical structures in an engineering description; it's the natural way to organize systems with a high degree of complexity. Similarly for software, the set of states and substates is most usefully represented in terms of complex hierarchies of attributes.

The use of hierarchies in VisiSoft languages is most apparent in the structure of resources. An example of VisiSoft hierarchical data structures is shown in Figure 11-1, illustrating code from a “resource”. A “resource” is a collection of data descriptions organized hierarchically. These generalized hierarchical data structures support the direct representation of a physical system's natural hierarchy. By using level numbers, the syntax of the resource language encourages the grouping and structuring of data.

RESOURCE NAME: TRANSCEIVER			
TRANSCEIVER_INSTANCES			
1	TRANSMITTER	INDEX	
1	RECEIVER	INDEX	
GENERAL_PARAMETERS			
1	TRANSMITTER_POWER	REAL	INITIAL_VALUE 100
1	RECEIVER_THRESHOLD	REAL	
RADIO QUANTITY(500)			
1	TRANSCEIVER	STATUS	TRANSMITTING RECEIVING IDLE OFF
1	LOCATION		
2	X_POSITION	REAL	
2	Y_POSITION	REAL	
2	ELEVATION	REAL	
1	ANTENNA_HEIGHT	REAL	
1	ANTENNA_GAIN	REAL	
RECEIVER_CONNECTIVITY_VECTOR QUANTITY(500)			
1	POWER_AT_RECEIVER	REAL	
1	TOTAL_NOISE_POWER	REAL	
1	CONNECTIVITY_MATRIX	QUANTITY(500)	
2	PROPAGATION_LOSSES		
3	TERRAIN_LOSS	REAL	
3	FOLIAGE_LOSS	REAL	
3	TOTAL_LOSS	REAL	
2	SIGNAL_POWER	REAL	
2	SIGNAL_TO_NOISE_RATIO	REAL	
2	LINK_DELAY	REAL	
2	LINK	STATUS	GOOD FAIR POOR
TRANSCEIVER_RULES			
1	TRANSCEIVER_PROCESS	RULES	TRANSMISSION RECEPTION TURN_ON_TRANSCEIVER TURN_OFF_TRANSCEIVER

07/13/06

Figure 11-1. Example of a hierarchical attribute structure of a Resource.

The resource TRANSCEIVER itself is a hierarchical data structure that can be moved as a single entity with one instruction. As a simple experiment, try writing the equivalent of TRANSCEIVER in C or C++. Then try using TOTAL_LOSS in an arithmetic statement.

SIMPLE STRUCTURES

For declarative aspects of the resource language, consider the VisiSoft resource of Figure 11-2. Here the basic structure of a text message is given along with two specific message formats that define their content. Level numbers indicate structure. Fields within a field are given greater level numbers, for example `HEADER` under `FORMAT_A` contains three fields:

```
PRIORITY  ORIGIN  DESTINATION
```

```
RESOURCE NAME: MESSAGE_FORMATS

MESSAGE
  1 SYNC_CODE          CHARACTER 6
                        ALIAS  VALID      VALUE '101010',
                        '010101'
  1 TYPE                STATUS FORMAT_A
                        FORMAT_B
  1 CONTENT             CHARACTER 46

FORMAT_A  REDEFINES MESSAGE
  1 PAD                CHARACTER 14
  1 HEADER
    2 PRIORITY          STATUS FLASH
                        IMMEDIATE
                        ROUTINE
    2 ORIGIN            INDEX
    2 DESTINATION       INDEX
                        ALIAS  BROADCAST  VALUE 0
  1 BODY
    2 LENGTH            INTEGER
  1 TRAILER
    2 MESSAGE_NUMBER    INTEGER
    2 TIME_SENT         REAL
    2 TIME_RECEIVED     REAL
    2 ACKNOWLEDGMENT    STATUS RECEIVED
                        NOT_RECEIVED
    2 LAST_SYMBOL       CHARACTER 2
                        ALIAS  TERMINATOR VALUE '\\', '/', '<<', '>>'

FORMAT_B  REDEFINES MESSAGE
  1 PAD                CHARACTER 14
  1 HEADER
    2 SOURCE            INDEX
    2 SINK              INDEX
  1 BODY
    2 CONTENTS          CHARACTER 42

7/13/06
```

Figure 11-2 Code from a VisiSoft “resource”

The basic message contains 60 characters, 14 in the header and 46 in the body. An input message field defined simply as 60 characters (CHARACTER 60) can be moved directly to the top level attribute MESSAGE. The 60 character block is moved into memory directly. The FORMAT_A and FORMAT_B structures are templates that overlay the memory. This example cannot be replicated in C. Because the numeric fields do not reside on word boundaries, the compiler puts padding into the structure automatically. So C programmers typically move the individual fields, unless they have control over the organization of records being read off a database. But this is impractical, since a database administrator should not care about the quirkiness of a particular language. More importantly, C programmers looking at VisiSoft code typically remark that it looks “very inefficient”. By actual tests, one can gain an order of magnitude in speed when reading records from commercial databases using VisiSoft directly into a template of multiple fields.

If one tries to translate the structure in Figure 11-1 to C, one immediately realizes why such data structures are not used. If that is not convincing, try using one of the lowest level fields - the ones of interest - in a process statement. For example, in C, TIME_RECEIVED becomes

```
MESSAGE_FORMATS.FORMAT_A.TRAILER.TIME_RECEIVED
```

MOVING DATA

VisiSoft resource structures are independent of the *word length* of a particular machine. Most of today's machines have 4 byte words, but some designs have larger words (the CRAY has 8 byte words, the NEXT machine has 36 bit words). In VisiSoft, the developer can lay out a very complex hierarchical attribute structure that is convenient to describe the module without worrying about *word boundary alignment* as in typical programming languages. Machine words have no meaning in VisiSoft. What you see (in your attribute structure) is what you get (in memory), independent of the machine you are using!

The MOVE statement is used to move data or assign values to variables. The format of the value is adjusted to fit the receiving field. For example,

```
MOVE ATTENUATION_FACTOR TO STORED_NUMBER
```

where

```
ATTENUATION_FACTOR is REAL
STORED_NUMBER is DECIMAL 9(2).9(3)
```

will result in

<u>Attribute</u>	<u>Before move</u>	<u>After move</u>
ATTENUATION_FACTOR	3.276000E-1	3.276000E-1
STORED_NUMBER	undefined	00.327

When the receiving area of a MOVE statement is a *numeric* value or a *decimal* value,

- Decimal points will be aligned and digits of the sending number will be truncated at either end, as required by the size of the receiving area.
- When a numeric value is moved to a DECIMAL area, zeros are changed to spaces when zero suppression (Z) is specified.
- When the receiving area of a MOVE statement is a CHARACTER value, data is aligned on the left and is either truncated at the right or filled with spaces to match the size of the receiving area.

As a result, formatting data becomes especially easy, and done in a way that is easy to understand.

Hierarchical Group Moves

The MOVE statement is particularly useful when assigning values to a structure, which may contain a mixture of attribute types.

A move in which one or both of the sending and receiving attributes are group attributes is called a *group move*. A group move is treated as a data move, without consideration for the elementary attributes contained within either the sending or receiving attributes. It is also possible to specify an entire resource as the sending or receiving area of a group move.

The ability to move a complete hierarchical structure, or any substructure within a hierarchy with a simple MOVE statement is most important in modeling the flow of information in a system. A good example is moving messages or message elements around in a communication system. These group moves are executed very easily, since one need only refer to the attribute name of the highest level group to be moved, and not worry about its size or structure. The modeler need only insure that the receiving structure is organized in a way that receives the data being moved into it. This ability to do hierarchical structure moves is key to the machine independent properties of the VisiSoft language.

SIMPLIFIED NAMING

VisiSoft has a generous facility for naming, allowing simpler, more understandable code. We begin with a brief review of conventional practices in other languages.

Most languages have some kind of facility for record structures. For example, in C++ we may have the following:

```
struct Message {  
    char A [message_size];  
    int first;  
    int last;  
};
```

or

```
class Message {public:  
    char A [message_size];  
    int first;  
    int last;  
};
```

In either case we can declare an object of this particular structure as follows:

```
Message My_Message;
```

To reference a component of a structure, we refer to the name of the structured object followed by the name of its component, for example,

```
My_Message.first
```

In general, a class of structure will have components, some of which may in turn be other structures. For any practical problem with interesting data, complex data structures are common. This gives rise to a sequence of names to refer to a component of a structure, as in

```
Name1.Name2.Name3.Item
```

With object-oriented approaches, access to a component of a structure is sometimes done with a method call rather than a direct reference to an individual data item, for example,

```
N = SystemParams.getDefault().getMemorySize();
```

Again we may have a cascade of names in order to designate a particular item of data in a structure. In larger programs, this nesting of structures may be several levels deep, with the result that many references become long and complex. Lengthy references to items in a data structure result from the requirement that all of the ancestors be referenced, from the outer level, in order to refer to the item.

In particular, having defined a hierarchical structure in C, C++, or Java, one must qualify each attribute (data item) with all of the names up the hierarchical chain, independent of whether that name is unique. For example, when using an attribute at three levels down in a structure, one must write all of the three names - separated by decimal points - to qualify the fourth level attribute, even though it is unique. Something like

```

if BUILDING_DESCRIPTION.ENTRANCES.FRONT.DOOR == OPEN
or BUILDING_DESCRIPTION.ENTRANCES.BACK.DOOR == OPEN
    MAKE_ENTRY()
else if BUILDING_DESCRIPTION.ENTRANCES.BACK.WINDOW == OPEN
    CHECK_ENTRY()
else ...

```

This makes lines long and unreadable.

Although these operations are sequential in nature, just understanding the reference to a single item attribute may be difficult for a reader other than the author to discern. (More false job security?)

The *VisiSoft* Solution

VisiSoft tackles this general problem in several ways, each devoted to keeping the naming as simple as possible while retaining great clarity and understandability of the algorithms. The first principle has already been discussed; that is:

- A process must be directly connected, architecturally, to all resources that contain data referenced by that process (there is no global data).

Put another way, instructions must be connected to the data segments needed by the instructions, and need not be connected to any other resources or data segments in the program. This greatly simplifies naming, for the names contained in the connected resources are directly visible without regard to any qualification of its parent or other ancestors.

Of course, as in any language, other connected resources compete for names. This brings us to the second rule in VisiSoft which is:

- Any name can be directly referenced as long as it is uniquely qualified.

Knowing that only connected resources compete for names, consider the following:

```

1  ENTRANCES
2  FRONT
   3  DOOR      STATUS      OPEN CLOSED LOCKED
   3  GARAGE    STATUS      OPEN CLOSED
2  BACK
   3  DOOR      STATUS      OPEN CLOSED LOCKED
   3  WINDOW    STATUS      OPEN CLOSED

```

Reuse of names that refer to different attributes is allowed provided the intended use can be uniquely resolved. Here, GARAGE and WINDOW can be directly referenced, without qualification. On the other hand, the use of the name DOOR to mean FRONT DOOR or BACK DOOR is resolved by adding the qualifier FRONT or BACK - that's it!

A conditional statement using the above resource can thus be written as follows:

```
IF GARAGE IS OPEN OR FRONT DOOR IS OPEN
    EXECUTE MAKE_ENTRY
ELSE IF WINDOW IS OPEN
    EXECUTE CHECK_ENTRY
ELSE ...
```

In the case of OPEN or CLOSED, reuse of STATUS names is qualified automatically by the particular status attribute FRONT DOOR or BACK DOOR.

Generally, any name in VisiSoft can be directly referenced as long as it is unique. Names are reusable within the same resource or over multiple resources. Reuse of names in a VisiSoft process requires qualification only to the extent sufficient to insure uniqueness of the referenced item. When a name appears in two resources, this may be accomplished just using the resource name - no matter what the level of the referenced item. All of which is a great step towards the simplicity, brevity, and understandability of the code.

Two other considerations are appropriate to this discussion. One is the use of architecture to ensure independence of data:

- Resources connecting two processes should only contain those attributes that must be shared between the processes.
- Attributes used by a single process, such as temporary attributes, pointers, or counters, should be contained in a resource dedicated to that process.

Another consideration is the practice of good naming conventions. *Names are a major contributor to understandability.*

- Attributes should not be used for more than one purpose, even in a dedicated resource.

Then names can be dedicated to one use and need not be generic, e.g., I, J, K, etc. Understandability increases significantly when names are meaningful, e.g., RECORD_COUNT, CHARACTER_POINTER, etc.

The time it takes to think of a name that clearly represents what the attribute itself is representing is a great investment in the future reuse of resources and particularly processes that use it in complex algorithms. It should go without saying that the time to type it is inconsequential.

STATUS ATTRIBUTES AND ALIAS CLAUSES

The STATUS clause is used to indicate each of the allowed states that a variable may assume during execution. STATUS attributes correspond to enumeration types in other languages. By defining a STATUS attribute, the programmer can set the state of an attribute to a predefined named state, then test to see if that attribute is set to a predefined named state. For example, we may have

```
TRANSCEIVER      STATUS TRANSMITTING
                  RECEIVING
PROBABILITY      STATUS LOW, MEDIUM, HIGH
```

This significantly improves the understandability of complex conditional rules, for example,

```
IF TRANSCEIVER IS RECEIVING . . .
```

while significantly reducing the chances for a logic error.

The ALIAS clause in VisiSoft extends the idea of status attributes to a wider range of applications. An ALIAS clause enables one or more values to be identified by a single identifier called the *alias name*. The ALIAS clause may be used along with a CHARACTER, DECIMAL, INTEGER, INDEX, REAL, or DREAL attribute.

The list of numeric or nonnumeric literals, separated by commas, specify the group of values which are to be associated with the alias name. More than one ALIAS clause may be specified for a variable. Some examples are

```
INPUT_MESSAGE
  1 LEAD_CHARACTER          CHAR 1
    ALIAS CONTROL_CHAR     VALUE 'S', 'R'
    ALIAS DELIMITER        VALUE '.', ',', ';', ':'
  1 MESSAGE_TEXT          CHAR 78
  1 LAST_DIGIT            INDEX
    ALIAS TERMINATOR       VALUE 0,9
```

Both STATUS values and ALIAS names can add to the understandability of a program. Thus we can have eminently readable statements such as

```
IF TIME_OF_DAY IS NOON
    SET RECEPTION_PROBABILITY TO HIGH

IF LEAD_CHARACTER IS A DELIMITER . . .

IF LAST_DIGIT IS NOT A TERMINATOR . . .
```

PROCESSING TABLES

Arrays become complex tables specified with a QUANTITY clause. For example,

```
MESSAGE_BUFFER
  1 MESSAGE QUANTITY(20)
  2 MESSAGE_HEADER
  3 MESSAGE_TYPE CHAR 8
  3 MESSAGE_PRIORITY STATUS LOW
                                MEDIUM
                                HIGH
  2 MESSAGE_BODY CHAR 68
```

Here MESSAGE_BUFFER contains 20 messages.

Processing tables of data is an important part of almost any large-scale application. The SEARCH table statement provides for automatic searching of tables over some or all indices, and execution of a rule when the specified table conditions are found to be true.

As an example, consider the following structure:

```
NUMBER_OF_TRANSCEIVERS INDEX
RECEIVER INDEX
TRANSCIEVER INDEX

LINK_CONNECTIVITY_VECTOR QUANTITY(500)
  1 CONNECTIVITY_MATRIX QUANTITY(500)
  2 PROPAGATION_LOSS REAL
  2 SIGNAL_TO_NOISE_RATIO REAL
  2 LINK STATUS GOOD FAIR POOR
```

To SEARCH this two-dimensional table executing TRANSMISSION for every LINK that is GOOD, one can use the following statement:

```
SEARCH CONNECTIVITY_MATRIX OVER RECEIVER, AND TRANSMITTER
EXECUTING TRANSMISSION
WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

To limit the search range to NUMBER_OF_TRANSCEIVERS instead of covering the 500 by 500 range, one would write the following:

```
SEARCH CONNECTIVITY_MATRIX
OVER RECEIVER TO NUMBER_OF_TRANSCEIVERS
AND TRANSMITTER TO NUMBER_OF_TRANSCEIVERS
EXECUTING TRANSMISSION
WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

To search the LINK_CONNECTIVITY_VECTOR to find the good links to a particular RECEIVER over the same range of TRANSMITTERs, one would write the following:

```
RECEIVER = SELECTED_RADIO
SEARCH LINK_CONNECTIVITY_VECTOR
  OVER TRANSMITTER TO NUMBER_OF_TRANSCEIVERS
  EXECUTING TRANSMISSION
  WHEN LINK(RECEIVER, TRANSMITTER) IS GOOD
```

This is a powerful feature for searching databases or parsing character strings. Much of the detailed and sometimes complex algorithms for table handling are done automatically and conveniently.



Chapter 12. Processes and Rule Structures

Almost every popular language includes some variant of if-then-else structures, do-while loops, and usually, some variant of a switch or case statement. These structures lead to a static understanding of control and are reasonably well behaved. In fact, these structures are so familiar and widely used that they go unquestioned.

Nevertheless, it is our perception that the way these control structures are defined leads to a major impediment to understandability. We believe that there are technical difficulties with these structures, and that additional elements need to be considered to achieve good understandability of code.

VisiSoft significantly improves understandability when dealing with complex algorithms, conditional statements, and repetition. The elegant solution implemented in VisiSoft is a clear departure and improvement over conventional control structures, implementing the one-in one-out structure suggested by Mills [66]. The VisiSoft approach is similar to that devised for COBOL, a language known for its readability. However, it eliminates two severe problems in the COBOL approach.

PROCESSES, RULES, AND STATEMENTS

The executable aspects of a software system in VisiSoft are embodied in a construct called a “process”. A process is a collection of executable statements organized in a hierarchical structure. A VisiSoft process can be invoked from any other process by using a CALL statement:

CALL process_name

Here control is immediately transferred to the called process.

Within a process, groups of statements are organized into “rules”. A rule is a named sequence of statements invoked by an EXECUTE statement. A process is thus defined as follows:

- A process consists of one or more rules, each with a unique name.
- Each rule name must appear on a separate line (starting in column 1) followed by one or more statements (starting in column 5 or beyond) that make up the rule.
- Each statement must begin on a new line, but can extend over many lines.

An example of a VisiSoft process is shown in Figure 12-1. The combination of statements and rules in a process form a logical structure of hierarchical levels.

Controlling Complexity With Rule Hierarchies

To control the complexity of highly conditional algorithms, a VisiSoft process can contain a hierarchy of rules. The hierarchy of rules is controlled through a simple *one-in, one-out* control structure, embodied in the EXECUTE statement. This statement allows the designer to deal with rules that are at an "equal level" in the hierarchy of logical operations, without resorting to the dangers of nested control structures.

When a process is invoked, the first rule is executed first, starting with the first statement. Other rules within this process may be executed by using an EXECUTE statement. Figure 12-2 shows an example of the process PLACE_CALL, which has five rules. A process terminates once the last statement in the first rule is performed.

What this means is that each process has a top-level rule (e.g., RULE_1), whose statements are executed in order. When the statements in this first rule have been executed, control returns to the calling process. Any rule may contain EXECUTE statements that invoke other rules within the process.

```

PLACE_CALL                                     level 1
  IF CLOCK_TIME IS GREATER THAN ONE_HOUR
    STOP.
  IF ACTIVITY(SOURCE) IS WAITING_TO_CALL
    EXECUTE ATTEMPT_CALL
  ELSE EXECUTE RETRY_LATER.

-----

ATTEMPT_CALL                                   level 2
  INCREMENT CALLS_ATTEMPTED
  IF LINES_IN_USE(OFFICE(SOURCE)) ARE LESS THAN
    LINES_IN_OFFICE(OFFICE(SOURCE)) THEN
    EXECUTE MAKE_CALL
  ELSE EXECUTE BLOCK_CALL.

RETRY_LATER
  SET ACTIVITY(SOURCE) TO RETRY_LATER
  CALL TERMINATE_CALL

-----

MAKE_CALL                                     level 3
  INCREMENT LINES_IN_USE(OFFICE(SOURCE))
  IF CALLERS_PLAN(SOURCE) IS PLACE_NEW_CALL
    SET PHONE_NUMBER TO UNKNOWN
    EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND.
  OFFICE_NUMBER = OFFICE(DESTINATION)
  CALL CONNECT_CALL

BLOCK_CALL
  INCREMENT CALLS_BLOCKED
  SET SIGNAL_TO_SUBSCRIBER TO BUSY
  MOVE 'BLOCKED AT SOURCE' TO CALL_STATE

-----

LOOK_UP_NUMBER                                level 4
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.

```

Figure 12-1. Example of the hierarchical rule structure of a process.

This dual structure has several advantages:

- a. Flow of control is always linear within a rule.
- b. At the end of a rule, control returns to the statement following the EXECUTE statement. This guarantees the 1-in, 1-out property.
- c. A process may contain one or more rules, each identified by a mnemonic name. This gives great flexibility in the number of conditional statements a process can support.
- d. There are no parameters passed to processes in VisiSoft, as is typical in conventional languages. All data is shared by resources.
- e. There is no nesting of IF statements.

Statements that are at the same logical level are all contained in the same spot. This makes them easier to build, and much easier to understand by someone other than the original author. The additional layer of hierarchy in a process allows the designer to partition complex algorithms that deal with the same attribute structures into isolated sets at similar levels in a logical hierarchy.

The example of Figure 12-1 involves at least 4 levels of control, yet is strikingly simple to understand. It also shows the ability to "push down" the complexity of rule sets into hierarchical logical levels. As a result, a process is typically somewhat larger than a "well written" C++ or Java function that are more the size of a rule. But it should be much more understandable, and will require many fewer comments, perhaps even none. Furthermore, a process with 20 rules may take a number of C++ or Java functions to implement.

SEPARATION OF CONTROL STRUCTURES FROM STATEMENT STRUCTURES

A *Process* defines the way a system transitions from state to state. A process is comprised of a set of *rules* that determine how the resources available to that process change depending upon their current state. (We must emphasize that the term *rule* as used here is different from its use in a rule-based language, e.g., PROLOG.)

Understandability of Complex Conditional Situations

One of the most important benefits of the VisiSoft conventions for rule structures is the handling of complex conditional situations. Consider the example in Figure 12-2. In particular consider

```
IF SYMBOL IS AN UNDERSCORE
OR SYMBOL IS A PERIOD
    EXECUTE CHECK_WORD_BLOCK
ELSE
    EXECUTE SCAN_FOR_SPECIAL_CASES.

IF STATEMENT IS A SPECIAL_CASE
    EXIT THIS RULE
ELSE ...
```

Here we see the equivalent of a case statement. But in VisiSoft, the statements that are contained within the case statement may be placed later in the process and given a name, in this case `CHECK_WORD_BLOCK` and `SCAN_FOR_SPECIAL_CASES`. This adds great clarity to the entire process, as we can read and understand the top level of control without getting involved in nested details that may be quite complex.

To simplify IF ... THEN ... ELSE chains, an EXIT THIS RULE statement allows one to exit a rule directly. This eliminates additional IF statements that must check a status attribute that has been changed above, simply by exiting the rule immediately after the change.

```

BUILD_WORD_BLOCKS
  ADD 1 TO SEARCH_INDEX
  MOVE INPUT_CHARACTER(SEARCH_INDEX) TO SYMBOL
  IF SEARCH_INDEX IS GREATER THAN 72
  AND SYMBOL IS NOT EQUAL TO SPACE
    MOVE '10390' TO ERROR_CODE
    EXECUTE REPORT_ERROR.

  IF SYMBOL IS AN UNDERSCORE
  OR SYMBOL IS A PERIOD
    EXECUTE CHECK_WORD_BLOCK
  ELSE
    EXECUTE SCAN_FOR_SPECIAL_CASES.

  IF WORD_BLOCK IS STARTED
    EXECUTE CHECK_WORD_BLOCK.

  IF SYMBOL IS NOT EQUAL TO SPACE
    MOVE SYMBOL TO LAST_NONBLANK_CHARACTER.

  IF SEARCH_INDEX IS EQUAL TO RECORD_SIZE
    SET WORD_STATE, COMPLETION_STATE TO COMPLETED.

CHECK_WORD_BLOCK
  SET SCAN_TYPE TO WORD
  MOVE DEBLOCK_WORD(WORD_INDEX) TO KEY_WORD_TABLE
  MOVE ZEROS TO CHARACTER_INDEX
  SET WORD_STATE TO BEGIN
  IF DEBLOCK_WORD(WORD_INDEX) IS NOT EQUAL TO SPACES
    SET LITERAL_TYPE TO NON_NUMERIC
    ADD 1 TO WORD_INDEX.

SCAN_FOR_SPECIAL_CASES
  IF .....
    STATEMENT_1
  ELSE IF .....
    STATEMENT_2

```

Figure 12-2. Un-nested Conditional Structures

Understandability of Loop Structures

A related and visible property of a VisiSoft process is the use of loop structures that isolate the body of the loop (the statements to be repeated) in a separate rule. Only the name rule is used within the control structure itself.

Thus we must write something like

```

EXECUTE LOOK_UP_NUMBER
  UNTIL PHONE_NUMBER IS FOUND

```

and place the body of the loop elsewhere

```

LOOK_UP_NUMBER
  DESTINATION = (TOTAL_SUBSCRIBERS * RANDOM) + 1
  IF DESTINATION IS NOT EQUAL TO SOURCE
    SET PHONE_NUMBER TO FOUND.
. . .

```

After the body of the loop LOOK_UP_NUMBER is executed, control automatically returns to the EXECUTE statement.

This is a powerful feature for clarity. For rather than a sequence of nested structures, we can again always read the control at a single level.

Logical Levels and Independence

Figures 12-1 and 12-2 also illustrate process structures that follow the rule for grouping hierarchical logical levels. Since the *logical levels are totally independent of position*, the process need not be organized this way, but in any manner the designer deems most understandable. Except for the first rule appearing first, the rest of the rules can be shuffled like a deck of cards.

Probably the largest benefit of the hierarchy of rule structures within processes is the understandability of complex conditional statements, and the ease with which one can add new conditions. These hierarchical structures support the direct representation of a physical system's natural flow of control.

NESTED CONTROL STRUCTURES IN CONVENTIONAL LANGUAGES

Nesting of control structures is a feature of virtually all conventional programming languages. For example, it is not uncommon to see

```

An if-statement
  containing an if-statement
    which contains a while-loop

```

Such an example by itself is not especially problematic, but does suggest the mental complexity of keep track of code with nested control. Moreover, the mental complexity increases as the length of the code and the length of nested sequences grows. It is not uncommon for single blocks of code to extend over more than one page

Things can get complex even without a nested loop. When there is nesting and the statements contained in the IF are of some length, the problem is getting a clear picture of the entire structure. When nested IF's cover many lines, it is hard to see what is going on.

As we get into larger or more complex situations, we mire ourselves in the complexity of logical flow. The program can eventually become unreadable. And this is often the accepted norm on software projects. Along these lines, we note that recursion is not allowed in VisiSoft.

Rule Pointers and Process Pointers

A nice step towards simplification and understandability of processes is the VisiSoft ability to assign the name of a rule or a process to a “rule pointer” or “process pointer”. Let us look at rules.

The RULE clause is used to define the allowed rule names that a pointer attribute can assume during execution. Consider:

```
NEXT_ACTION      RULE  INITIALIZE_NETWORK
                  START_TRANSMISSION
                  START_RECEPTION
                  DISCONNECT_CALL
```

Here, NEXT_ACTION is a Rule attributes.

By defining a RULE attribute, the modeler can execute a rule based on the value of the rule attribute. This can be simulated by a case statement in a conventional programming language. The value of the VisiSoft approach is the simplification on control flow. Meaningful names can be used for the rules and the rule-pointer, and the choice of action can be set when an appropriate condition is met.

The rule pointer will likely be set in a conditional statement prior to a point where the rule is to be executed, such as:

```
IF TRANSCEIVER(TRANSMITTER) IS TRANSMITTING
    SET NEXT_ACTION TO START_TRANSMISSION
ELSE IF TRANSCEIVER(RECEIVER) IS RECEIVING
    SET NEXT_ACTION TO START_RECEPTION
```

Then, at the point where the choice of rules is to be executed (that choice will already have been made as above), one merely EXECUTE's the rule pointer name.

```
EXECUTE NEXT_ACTION
```

Process Pointers

The PROCESS pointer clause is similar to the RULE pointer clause. It is used to define each of the allowed process names that a PROCESS pointer attribute can assume during execution. It is used to support the PROCESS pointer version of the CALL statement for executing processes.

The mechanism is almost identical to rule pointers. For example,

```
NEXT_PROCESS  PROCESS  COMPUTE_TIMERS
                DRAW_TERRAIN
                COMPUTE_MEASURES
. . .
IF INPUT_OPTION IS INITIATE
    SET NEXT_PROCESS TO COMPUTE_TIMERS
ELSE IF INPUT_OPTION IS CALCULATE
    SET NEXT_PROCESS TO DRAW_TERRAIN
```

Then, at the point where the choice of process is to be called, (that choice will already have been made as above) one merely CALL's the process pointer name.

```
CALL NEXT_PROCESS
```

INTERTASK COMMUNICATIONS AND CONTROL

The next example provides for two very simple interactive tasks, each sending messages to the other. The messages are input via the keyboards of each task, and appear on the screen of the other task. This is done using separate windows controlled by separate tasks running concurrently under the VisiSoft Run-Time Monitor. Figure 12-3 below shows the architecture of this simple example. The implementation follows in Figure 12-4.

The session starts by the user running task 1, which automatically opens a window. Task 1 immediately starts Task 2, with a window, and suspends itself. The very first message of the session, 'ASK A QUESTION', is put on the screen of task 2 by initialization. From then on, the conversation proceeds with the keyboard entry being put into CONVERSATION_BUFFER, an intertask resource. The task that accepts input from the keyboard then resumes the other task and suspends itself. When a task is resumed, it displays the message in CONVERSATION_BUFFER upon the screen, accepts the next input from the keyboard, putting it into the CONVERSATION_BUFFER, resumes the other task and suspends itself. This continues until one of the keyboard entries is STOP.

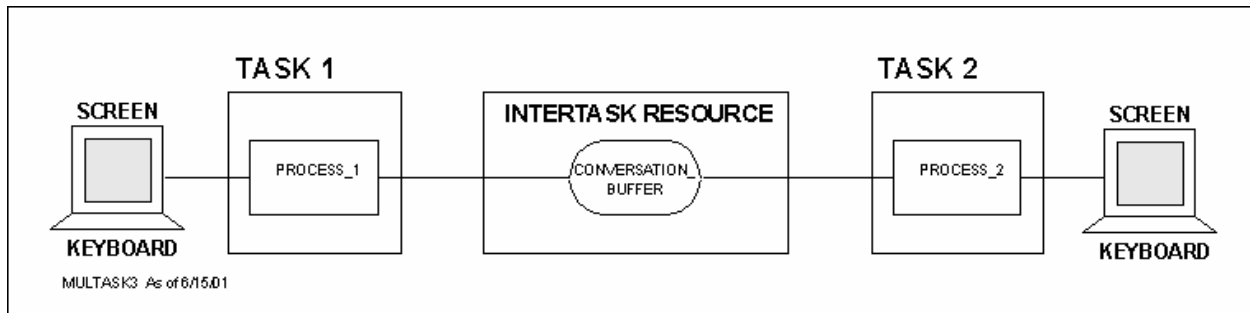


Figure 12-3 Architecture of a real-time intertask communications example.


```

Intertask Resource:

CONVERSATION_DATA
    1    ANSWER                CHAR  4
                                ALIAS  STOP    VALUE 'STOP'
    1    REST                  CHAR 60

Task 1

PROCESS_1
    START TASK_2 WITH WINDOW
    SUSPEND TASK_1
    EXECUTE ANSWER_A_QUESTION
        UNTIL ANSWER IS STOP
    TERMINATE THIS TASK

ANSWER_A_QUESTION
    DISPLAY CONVERSATION_BUFFER
    ACCEPT CONVERSATION_BUFFER
    RESUME TASK_2
    SUSPEND TASK_1

Control Specification for Task 1

*CONTROL SECTION
    TITLE, EXAMPLE OF INTER-TASK COMMUNICATIONS & CONTROL
    LEAD_PROCESS IS PROCESS_1
*END

Task 2

PROCESS_2
    MOVE 'ASK A QUESTION' TO CONVERSATION_BUFFER
    EXECUTE ASK_A_QUESTION
        UNTIL ANSWER IS STOP
    RESUME TASK_1

ASK_A_QUESTION
    DISPLAY CONVERSATION_BUFFER
    ACCEPT CONVERSATION_BUFFER
    RESUME TASK_1
    SUSPEND TASK_2

Control Specification for Task 2

*CONTROL SECTION
    TITLE, EXAMPLE OF INTER-TASK COMMUNICATIONS & CONTROL
    LEAD_PROCESS IS PROCESS_2
*END

```

Figure 12-4. Elements of a simple two task example.

Anyone who has worked with intertask communications and control in UNIX, referred to as Inter-Process Communication (IPC), will testify to the level of difficulty involved in creating the little example above. All of the effort of setting up and managing shared memory control blocks, shared memory areas, and the difficulties of putting processes to sleep and sending signals to wake them up is done for the user, behind the scenes, by VisiSoft. These VisiSoft features ease the programming of real-time communications and control applications.



CHAPTER 13 CONTROL SPECIFICATIONS

In contemporary software environments, there are facilities to put together the components of a project. Typically, a project will require different kinds of resources, e.g. a compiler (perhaps a debugging compiler), libraries, access to operating system routines, files, access to directories, and so forth. On some systems, there is a specific language to control these aspects, generally known as a script or Makefile.

VisiSoft handles this issue in an elegant way that is independent of both the machine and operating system. This is through a separate high level language known as the “Control Specification” language. An example is shown in Figure 13-1.

The Control Specification is eminently readable and organized. It contains a sequence of labeled sections. Each section specifies some property of the environment. The notation used for the syntax is based on a simplified English-like format.

```

CONTROL SECTION
  TITLE, SIMPLE TELEPHONE SYSTEM
  LEAD_PROCESS IS INITIALIZE_NETWORK

LIBRARY SECTION
  C:/S/LIBS/GENERAL
  C:/S/LIBS/RTG_DRAW

GRAPHICS SECTION
  ACTIVATE GRAPHICS
  WORLD_SPACE LOWER_LEFT = (0, 0),
                UPPER_RIGHT = (1280, 1024)
  NVS/BDS = 1.0
  INITIAL_WINDOW LOWER_LEFT = (-100, -100), WIDTH = 1280

  ICON OFFICE_OUTLINE = OFFICE,          SCALE(1.0, 1.0)
  ICON MAN              = MAN,           SCALE(2.0, 2.0)
  ICON PHONE           = PHONE,         SCALE(1.0, 1.0)
  ICON TERMINAL        = TERMINAL,      SCALE(1.0, 1.0)
  ICON PBX_LINE_TERM   = PBX_LINE_TERM, SCALE(1.0, 1.0)

  . . .

  INST GENERATED_CALLS = THERMOMETER_VERTICAL,
                        LOW 0, HIGH 400, INITIAL_VALUE 0, COLOR BLUE
  INST BLOCKED_CALLS   = THERMOMETER_VERTICAL,
                        LOW 0, HIGH 400, INITIAL_VALUE 0, COLOR BLUE

  . . .

  OVERLAY 3 = DRAW_SWITCH IN PHONE BACKGRND
             AT 0,0, SCALE 1, 1, MENU SWITCH
             COLOR BACK_BLUE
             ***COLOR BACK_WHITE
  OVERLAY 4 = DRAW_LABELS IN PHONE BACKGRND
             AT 0,0, SCALE 1, 1, MENU LABELS
             COLOR BACK_WHITE

  RTG_EVENT_HANDLER INTERACTIVE_SCENARIO

DATABASE INPUTS
  ASSIGN SFI INPUT_DATA.SFI TO READ_SCENARIO_DATA

DATABASE OUTPUTS
  ASSIGN SFI OUTPUT_DATA.SFI TO OUTPUT_TEST_DATA

END

```

Figure 13-1. Sample task control specification.

Some of the items addressed in the Control Specification include:

- **LEAD_PROCESS** The process named as the LEAD_PROCESS is started when the task is executed.
- **TRACE** A debugging facility used to trace processes, rules, and produce trace output when one or more processes have been prepared with the one of the trace options on.
- **TIME PROFILE** Provides a histogram of the percentage of time spent in each process.
- **LIBRARY SECTION** This section allows the user to specify the paths and names of libraries to be used when preparing a task that uses library modules.
- **GRAPHICS SECTION** This section is used to invoke the VisiSoft Run-Time Graphics facilities. This section has numerous options for the user.
- **DATABASE INPUTS AND OUTPUTS** These sections may be used to reassign external files to an external resource, or to invoke the Standard File Interface (SFI) option for input data to a task.
- **MODEL SECTION** Listed here are models that contain processes to be started in a simulation (only applies to GSS).

STANDARD FILE INTERFACE (SFI)

When users want to change or look at data files, they typically want to use an editor or print the files as raw data. If users want to put the resulting data into a spreadsheet, (e.g. EXCEL or SAS) for data analysis or plotting, or if they wish to create a readable report, they must do considerable work. The amount of time consumed is high compared to what it takes to understand and use a standardized file input and output system.

Ideally, one would like to have standard interfaces to readily available database management packages, e.g., Oracle, ACCESS, DB-2, etc., as well as spreadsheets, e.g., EXCEL and LOTUS, or statistical analysis packages, e.g., SAS and SPSS. This is why a number of users developed the Standard File Interface (SFI) formats. The SFI approach greatly simplifies reading and writing large sequential data files.

There are a number of facets to be understood in order for SFI to be appreciated. These include creation of the raw data files, editing of input data, and providing for standard file input to, and output from, a simulation so that users do not have to build data input and output modules for each file. SFI also provides for standard reporting facilities that take care of header information and page counting.

Most important, direct interfaces to database management systems for data entry and management, and to spreadsheets and statistical packages for data analysis and plotting is a necessary requirement today. All of these considerations are addressed with SFI.

All SFI files must contain one format record for each field in the data records. These are used to automatically recognize the data element names and their formats on input and output files, and to send and accept data from EXCEL, LOTUS, DBase, and other formatted databases. In addition, standard reporting and plotting facilities can be used directly with SFI files because the format records contain all of the information to determine what a user wants to see. See Figure 13-2.

```

* HUB - SUBSCRIBER DEPLOYMENT FILE
*****
* SFI HEADER RECORD FOLLOWS
*
TERMINATOR = SPACE, SPACES = 1
*
*****
* SFI FORMAT RECORDS FOLLOW
*
NAME = HUB_ID                INTEGER
NAME = AREA_CODE             INTEGER
NAME = NUMBER_OF_SUBSCRIBERS INTEGER
NAME = SERVICE_TYPE          CHARACTER
* C = CAS, L = LAM, S = SAM, I = SAM-SI, A = ADMIN
NAME = CABKE_ID              INTEGER
NAME = SERVICE_FREQUENCY     FLOAT
NAME = DEST_FREQ             FLOAT
NAME = MEAN_INTERGEN_TIME    EXPO
NAME = GREETING_TIME         EXPO
*
*****
* DATA RECORDS FOLLOW THE DESCRIPTION BELOW
*
*HUB AC SUB S CID   SFR   DF      MIT      GRT
*
001 908  5 C  30  65.8  2.73  39.2E+03  -.56E5
001 908  0 L   8  65.8  2.73  39.2E+03  -.56E5
001 908  0 S   4  65.8  2.73  39.2E+03  -.56E5
001 908  0 I   8  65.8  2.73  39.2E+03  -.56E5
001 908  0 A   4  65.8  2.73  39.2E+03  -.56E5
001 201 10 C  30  65.8  2.73  39.2E+03  -.56E5
001 201  0 L   8  65.8  2.73  39.2E+03  -.56E5
001 201  0 S   4  65.8  2.73  39.2E+03  -.56E5
001 201  0 I   8  65.8  2.73  39.2E+03  -.56E5
001 201  0 A   4  65.8  2.73  39.2E+03  -.56E5
001 609  8 C  30  65.8  2.73  39.2E+03  -.56E5
001 609  0 L   8  65.8  2.73  39.2E+03  -.56E5
001 609  0 S   4  65.8  2.73  39.2E+03  -.56E5
001 609  0 I   8  65.8  2.73  39.2E+03  -.56E5
001 609  0 A   4  65.8  2.73  39.2E+03  -.56E5
002 215  9 C  30  65.8  2.73  39.2E+03  -.56E5
002 215  0 L   8  65.8  2.73  39.2E+03  -.56E5
002 215  0 S   4  65.8  2.73  39.2E+03  -.56E5
002 215  0 I   8  65.8  2.73  39.2E+03  -.56E5
002 215  0 A   4  65.8  2.73  39.2E+03  -.56E5
002 610  7 C  30  65.8  2.73  39.2E+03  -.56E5
002 610  0 L   8  65.8  2.73  39.2E+03  -.56E5
002 610  0 S   4  65.8  2.73  39.2E+03  -.56E5
002 610  0 I   8  65.8  2.73  39.2E+03  -.56E5
002 610  0 A   4  65.8  2.73  39.2E+03  -.56E5

```

Figure 13-2. Example of an SFI input file

Each SFI input file must be associated with a unique VisiSoft process. This is accomplished when building the architecture by connecting a file icon to a process and selecting the desired SFI file name.

SFI input processes automatically call the SFI file input subsystem to read the next record and move each field into the user specified attribute. The SFI input subsystem automatically performs the following functions:

- Opens and closes the input files
- Reads each record from a file
- Reads each field from a record
- Transforms and checks numeric fields
- Produces appropriate error messages

This facility is directed squarely at easing the burden on the programmer and system designer, thus another contribution to increased productivity.

SPECIFICATIONS FOR RUN-TIME GRAPHICS

VisiSoft greatly simplifies the use of graphics for viewing output during run-time. When using the Run-Time Graphics (RTG) facility, the designer must identify elements that will be used from the graphics library. RTG control specification statements are used to invoke the graphics facilities, set parameters, and identify the graphic objects within their respective libraries. Therefore, the control specification contains an additional section, known as the GRAPHICS section.

Figure 13-3 provides an illustration of an RTG graphics window. This example contains icons, lines, instruments, and backgrounds. In this figure, the SWITCH, PBXs, and OFFICES can be drawn as backgrounds. The men and telephones are examples of RTG ICONS. The four thermometer type bars on the right are examples of RTG INSTRUMENTS. The lines interconnecting the switch with PBXs and PBXs to telephones are examples of RTG LINES. The overall picture is an illustration of what may appear in the RTG graphics window during execution.

Various options exist for setting the world space, window size, and the relative size of elements to be shown on the screen. Symbols to be inserted in the graphics window, whether from the application or by interactive input, must also be defined in the control specification.

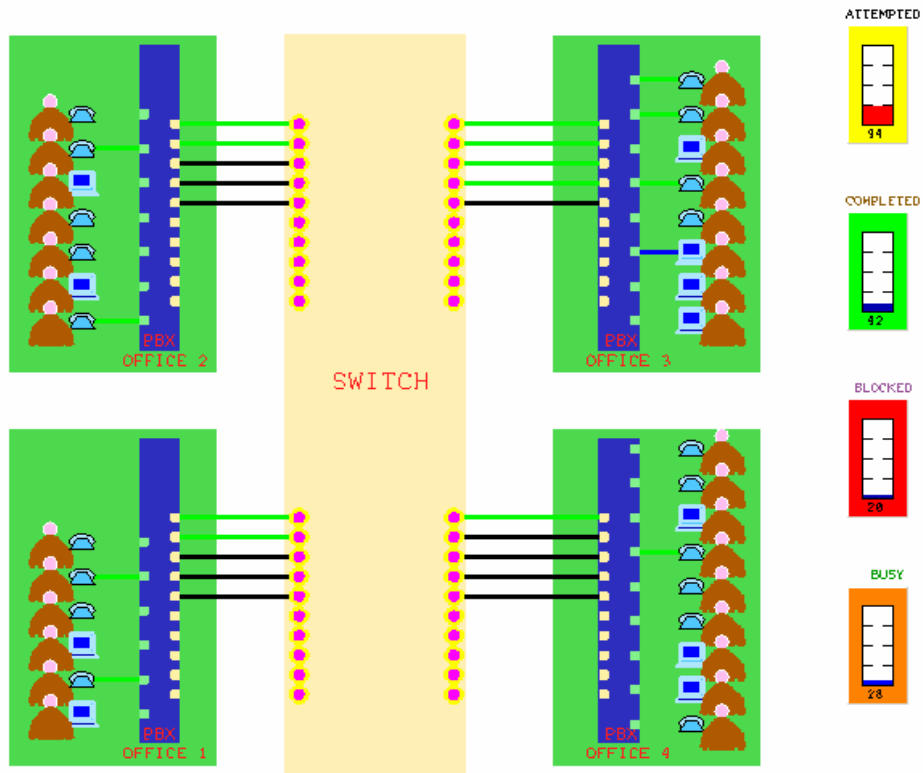


Figure 13-3. Illustration of icons, lines and instruments to represent network activity.

The Graphics Section

The GRAPHICS SECTION is used to define the graphics library elements to be available at run-time as well as the process that is invoked whenever an interactive input event occurs. At this stage the modeler defines the graphic symbols and their attributes. The following key identifiers are used:

ICON	- Defines Icons
INST	- Defines Instruments
LINE	- Defines Lines (connectivity links)
OVERLAY	- Defines the Background Overlays
RTG_EVENT_HANDLER	- Defines the Automatic Event Handling Process

Activating Graphics

Two separate modes of RTG graphics are available, ORTHO and PERSPECTIVE. ORTHO refers to a 2-Dimensional mode of operation for RTG, which provides a view looking in the negative Z direction. PERSPECTIVE refers to a 3-Dimensional mode of operation for RTG, which allows the user to view objects from any *viewpoint* in space toward a specified *look-at point*, such as the origin. The format for the statement defining the RTG graphics mode of operation is shown below. The default mode of operation is ORTHO.

$$\text{ACTIVATE [GRAPHICS] } \left\{ \begin{array}{l} \text{[ORTHO]} \\ \text{[PERSPECTIVE]} \end{array} \right\}$$

World_Space Definition

If an application has run time graphics, users can define the “play-box” and VIEW POINT for 2D and 3D graphics scenes.

The “play-box” or WORLD SPACE is the box inside of which all of the action takes place. This box is defined by the two points (Xmin, Ymin, Zmin) and (Xmax, Ymax, Zmax). These two coordinates define the diagonal line that spans the play-box. For example, X could range from -3 to +7 miles. Z could range from 0 to 80,000 feet. The format for the statement defining this rectangle box is shown below.

```
WORLD_SPACE LOWER_LEFT = (Xmin, Ymin [, Zmin])
                UPPER_RIGHT = (Xmax, Ymax [, Zmax])
```

If the run time graphics mode is PERSPECTIVE (3D), an initial view vector can be specified. The initial view vector is defined by two points:

the viewer's *viewpoint* (viewer_x, viewer_y, viewer_z), and
the viewer's *look-at point* (lookat_x, lookat_y, lookat_z).

The user must also decide how large symbols should be in the world space. The scale factor is called the NVS / BDS ratio, and can be set by the user. The default is 1, i.e., the size as drawn of the original icon.

All foreground objects and background overlays are drawn over the RTG window background color. By default, the RTG window background is black. Users can choose different *colors* (e.g., white instead of black).

Icon Names

Icons are drawn and defined in a special VisiSoft facility known as the Icon Library Manager (ILM). Within an application, an icon is given an internal program name, which in turn is associated with a specific icon defined in the ILM.

```
ICON icon_name = icon_library_name

    [, SCALE scale_x, scale_y [, scale_z]]
    [, COLOR color]
    [, STYLE style]
    [, THICKNESS thickness]
```

When the x, y, or z scale factors are specified, all of the icons in a hierarchy are scaled as well as their relative distances. The values for color, line style, and line thickness only apply to *variable-property icons*. These must be created in the ILM with parts whose color has been set to the *variable color*. For example, we may have

```
ICON PHONE = TELEPHONE_03, COLOR RED, STYLE 1, THICKNESS 1
ICON OFFICE = OFFICE_10, COLOR LIGHT_GREEN, STYLE 1,
              THICKNESS 3
```

Instruments

Instruments are predefined VisiSoft objects that take on special properties similar to actual instruments. The values assigned to an instrument describe the default settings that the instrument is to assume unless otherwise explicitly stated in a process. For example, we may have

```
INST CALLS_GENERATED = THERMOMETER, LOW 0, HIGH 400,
                       INITIAL_VALUE 0
```

Background Overlays

Background overlays are separate user models created using VisiSoft and VSE draw libraries, or Open-GL directly for very special functions. To incorporate a background overlay in an application, it must be defined in the control specification.

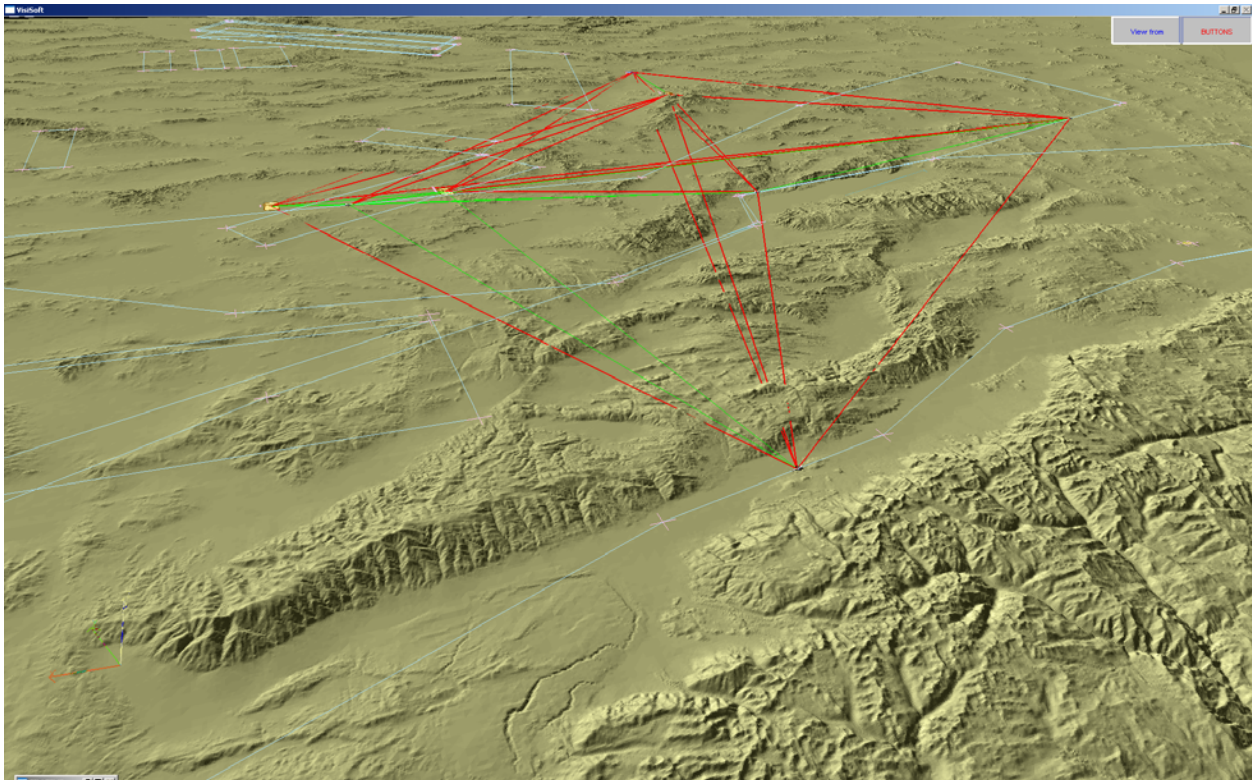
Each background overlay must be defined using a separate OVERLAY statement as shown in Figure 13-1. The sequence of names, i.e., *overlay_name*, *module_name*, and *library_name*, specifies the process *in a VSE library module* to be called to draw the overlay. The *menu_name* is placed in the background overlay list that can be used to interactively toggle any of 100 background overlays on or off.

By default, the origin - (0, 0, 0) point - of coordinates defined inside the overlay module is automatically registered relative to location (0, 0, 0) in the world space. The AT clause can be used to register the (0, 0, 0) point in the overlay to a different location in the world space. Likewise, the SCALE clause can be used to reconcile an overlay to the coordinates used for the world space. The default scale factor is 1.

The color or color ramps used by overlay draw routines can be changed in the control specification. Similarly, the coordinates of the light source must be provided when using 3-D shading. This coordinate is used by the overlay to determine the direction of the light source vector with respect to the *look-at* point mentioned above.

The user is responsible for creating background overlays. The RTG_DRAW library, available to the user directly from VSE, includes most of the utilities required to draw 2D or 3D background overlays.

Figure 13-1 illustrates the specification of some of the above features in the Graphics Section of a Control Specification. The RTG_EVENT_HANDLER clause identifies the process to be invoked automatically when a graphics event has occurred.



Chapter 14. Simple Examples

The above illustration is a screen shot from a simulation of multiple platforms moving and communicating in 3D terrain. The terrain is drawn using digitized terrain databases as an RTG background overlay. This chapter describes two examples of VisiSoft graphics using RTG to demonstrate the ease with which one can build graphical representations of system dynamics. One is a simple bouncing ball with a smiley face. The other is the game of TIC-TAC-TOE, where players could use their own computer on a network.

SMILEY - THE BOUNCING BALL

This is an example of a bouncing ball. It uses an RTG icon with a smiley face as the ball, and some simple equations to make the ball bounce in a somewhat realistic manner. As shown in Figure 14-1, the ball is pushed off a wall on the left, and bounces to the right, with diminishing height. This motion can be represented by the product of a sine wave and an exponential decay function, with parameters adjusted to suit the desired speed of motion. It is also simple to have the ball spinning for a bit of realism.

This example uses the Icon Library Manager to build the smiley icon, and uses a background overlay for the wall and floor. It also illustrates how one makes use of RTG to move icons against a background.

Defining The Problem

Figure 14-1 illustrates the desired motion of the ball, and the smiley face icon.

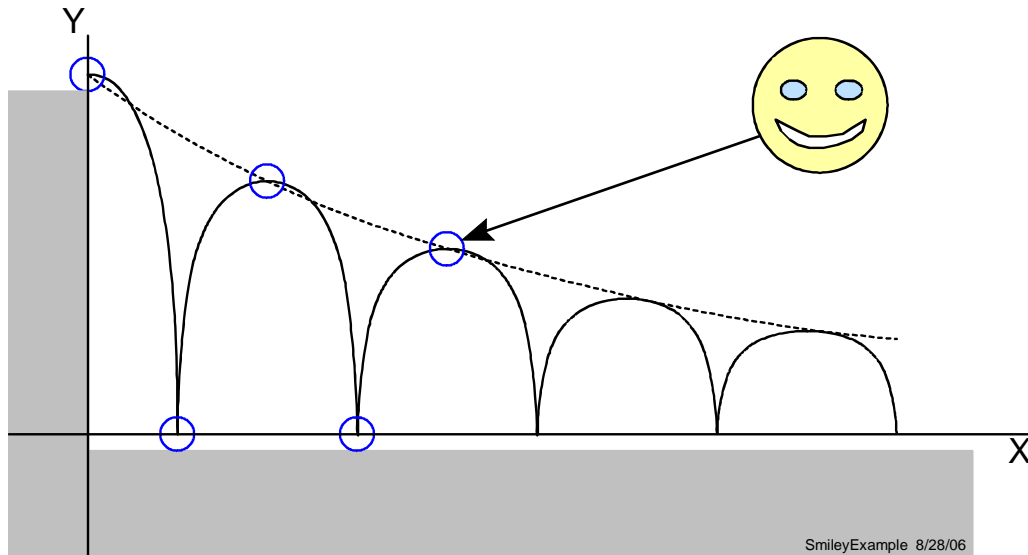


Figure 14-1. A ball is pushed off a wall and bounces away.

Motion is handled using the following simple set of equations as a function of time T :

$$X = K_1 \cdot T$$

$$Y = K_2 \cdot \text{ABS}[\text{SIN}(K_3 \cdot T)] \cdot e^{-aT}$$

$$\Phi = K_4 \cdot T$$

K_1 determines speed in the X direction. K_2 determines the amplitude of the sine wave which is modulated by the decaying exponential with time constant a . K_3 determines distance between bounces. Φ is the rotation angle of the ball, and K_4 determines speed of rotation.

Building The Icon

To build the icon, one must click on the ILM button in the VDE window. This brings up the Hierarchical ILM. Then one clicks on the Create Elementary button to get into the elementary ILM drawing board shown in Figure 14-2. Using the drawing tools on the left button bar, the face outline and eyes are formed using the ellipse, and the mouth is formed using the filled polygon. The grid is determined by parameters from the Status Bar, toggled by clicking on the Status button on the right side of the lower button bar. The icon is saved with the name SMILEY.

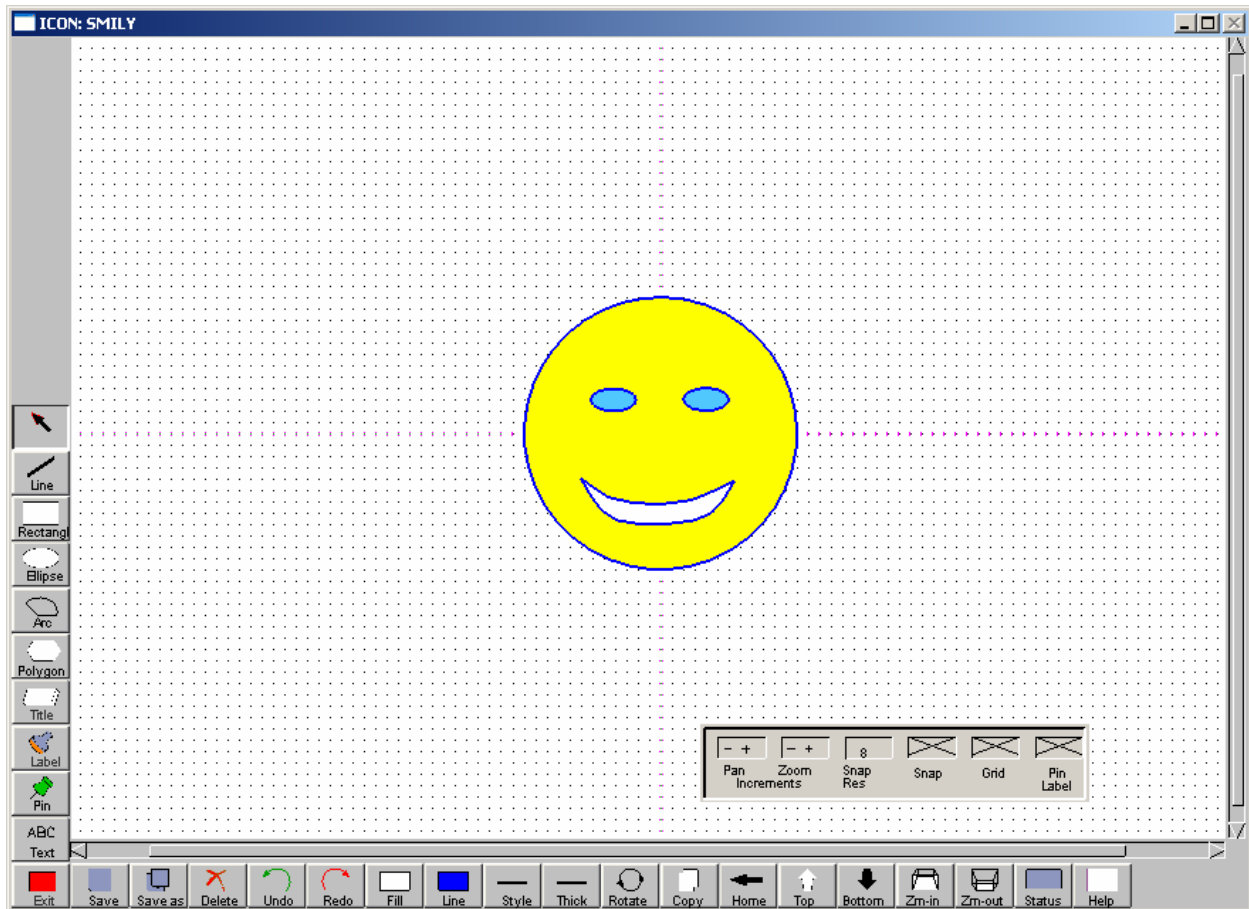


Figure 14-2. Building SMILEY using the Elementary Icon Drawing Board.

Building The Architecture

The basic architecture for bouncing smiley is shown in Figure 14-3. It requires only one resource and one process. These are both shown in their respective edit sessions in this figure. If we ignore the gray wall in Figure 14-4, this will run as is, bouncing the smiley ball, using a fairly simple control specification. The more complete one (with an overlay) is shown in Figure 14-6.

Building The Background Overlay

There are different ways to add in the wall. One could use an icon. But that gets redrawn in the foreground every time smiley moves. A better way is to use the background overlay facility within RTG, removing the requirement to redraw the background when only the foreground changes, as is the case here. To do this, the user creates a library module as shown in Figure 14-5. This process calls `DRAW_RECTANGLE` in `DRAW_MOD` in `RTG_DRAW`, a VSE library that provides the facilities of Open-GL without having to write C code.

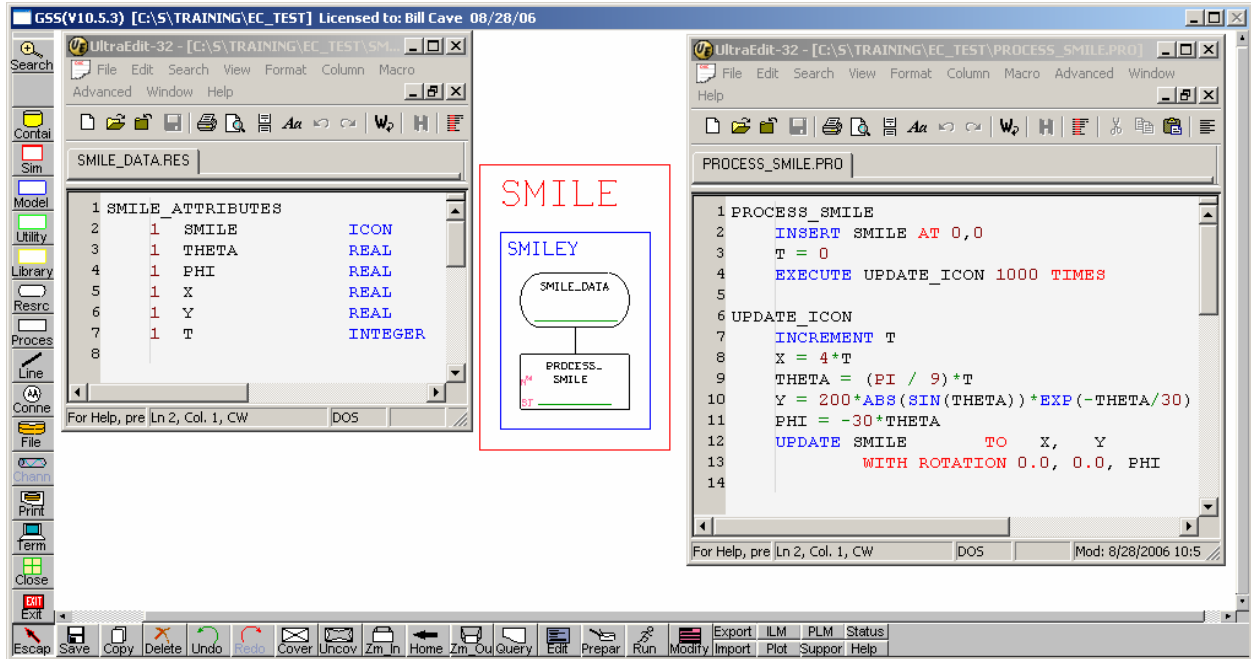


Figure 14-3. Architecture for bouncing SMILEY.

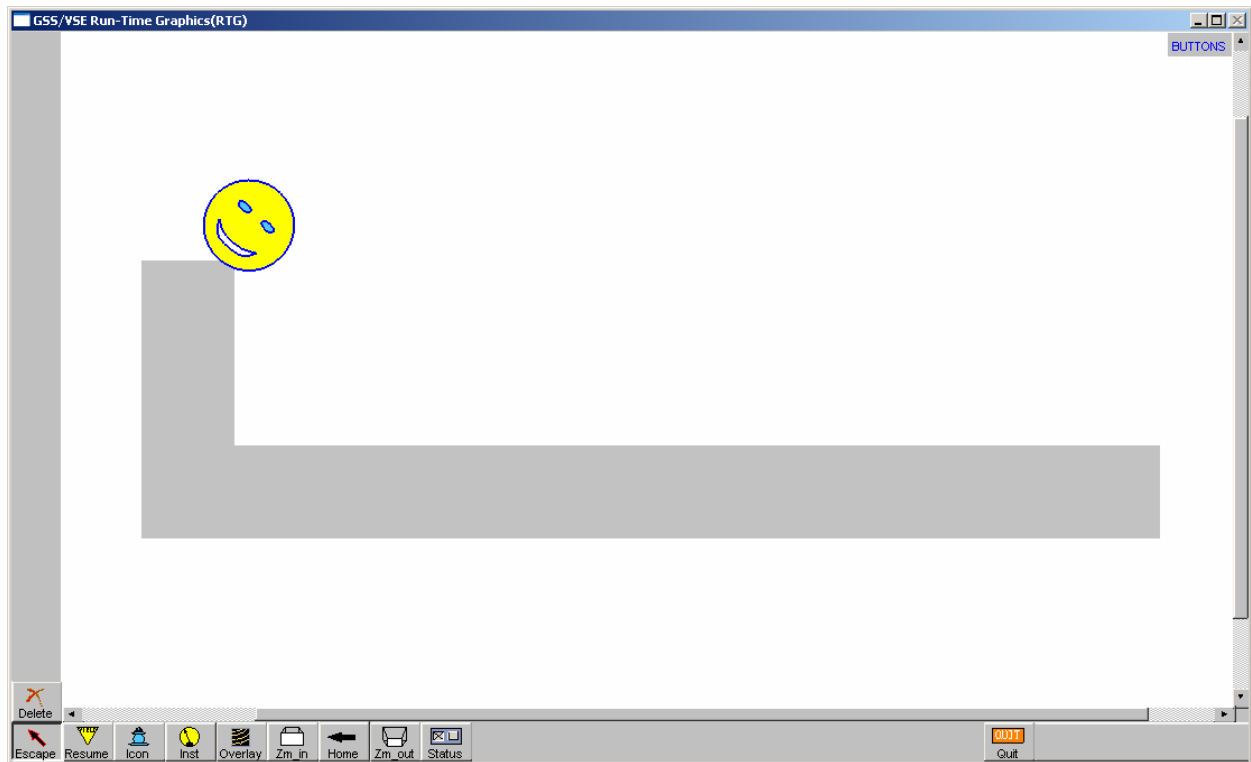


Figure 14-4. Running bouncing SMILEY.

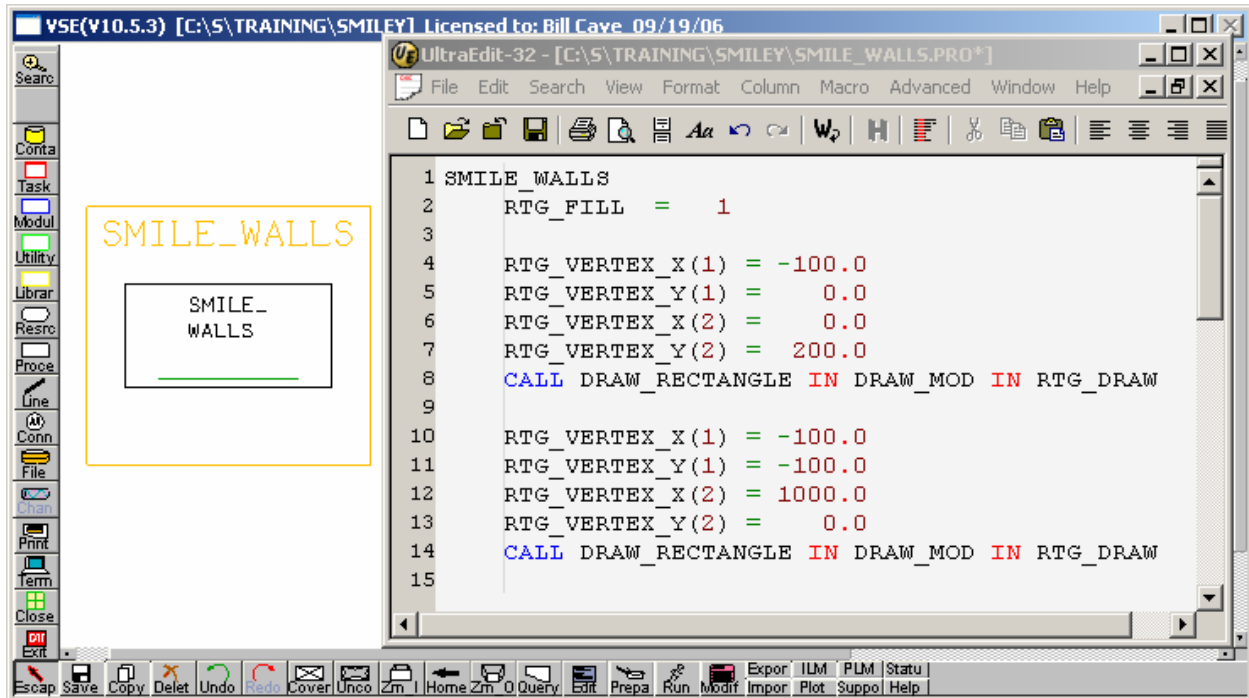


Figure 14-5. Architecture for SMILE_WALLS.

```

CONTROL SECTION
  TITLE, TEST OF SMILE
  LEAD_PROCESS IS PROCESS_SMILE

LIBRARY SECTION
  C:\S\LIBS\RTG_DRAW

GRAPHICS SECTION
  ACTIVATE GRAPHICS
  WORLD_SPACE LOWER_LEFT = (-500, -500), UPPER_RIGHT = (500, 500)
  BACKGROUND_COLOR = WHITE

  ICON SMILE = SMILEY, SCALE (0.5, 0.5)

  OVERLAY 1 = SMILE_WALLS IN SMILE_WALLS IN SMILE_LIBRARY
              MENU WALLS,
              AT (0.0, -50.0)
              COLOR GRAY

END

```

Figure 14-6. Control Specification for SMILEY.

To create the walls using the RTG_DRAW library, one merely sets the fill to 1 if the object being drawn is to be filled with a color. In this case, we are drawing two rectangles, so we must specify the lower-left and upper right vertices and call the DRAW_RECTANGLE routine in module DRAW_MOD.

Then, to make this work, we must add the LIBRARY SECTION into the Control Specification in Figure 14-6, and also specify the overlay module (as OVERLAY 1 here). We must provide the process_name, module_name, and library_name that we have created above. We must also provide a menu name if we want to turn it on and off, the point at which the object will be inserted, and the color of the object if filled.

To bring up the walls (overlay) automatically, we must add an INSERT OVERLAY 1 statement at the top of PROCESS_SMILE. Some minor changes are required to the formulas to start the smiley ball at the top of the wall. This is best done by setting $T = 3$ instead of 0 to start.

Building A Panel To Change Speed

Now let's put in a panel to change the speed interactively. We will use a slider bar such as shown in Figure 14-7. To do this, we must first add a panel resource to the architecture shown in Figure 14-7

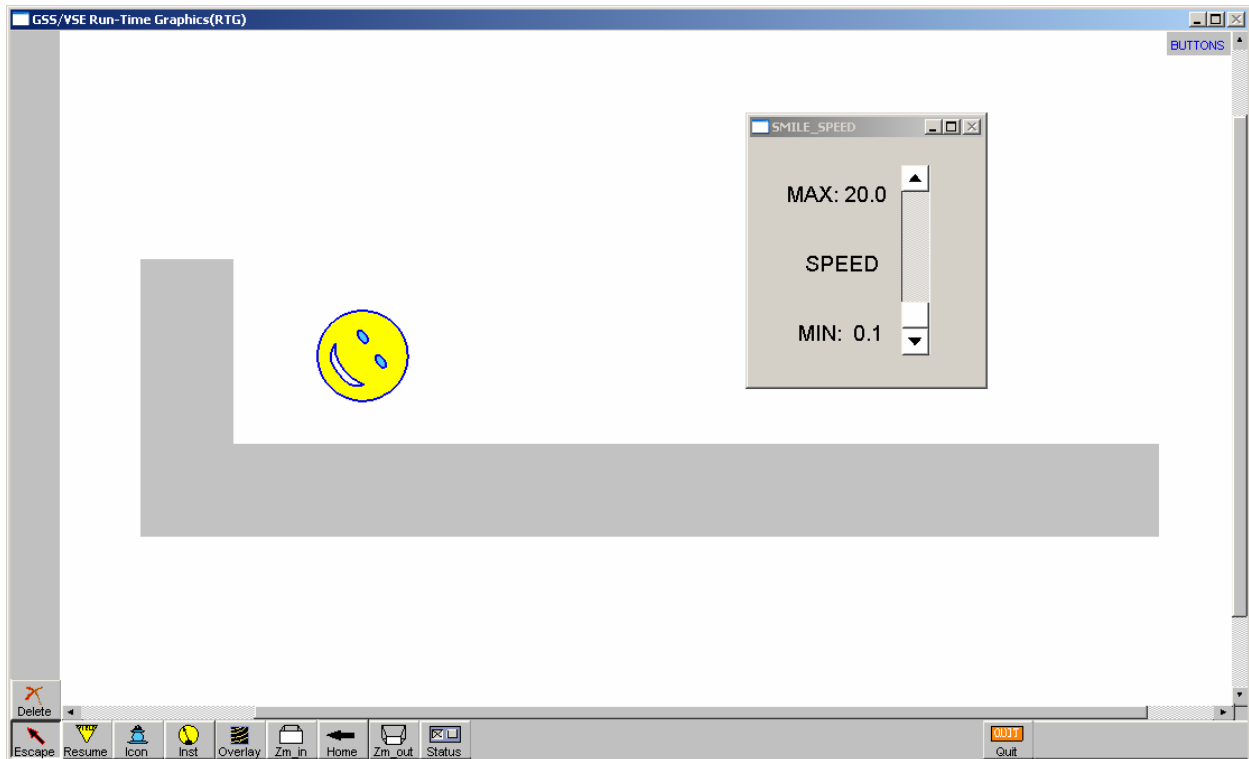


Figure 14-7. A slider bar for controlling SMILEY's speed.

The SMILE task architecture shown in Figure 14-8 has been augmented with a SMILE_SPEED resource at the bottom of module SMILEY. This resource holds the information for the slider bar shown during run time in Figure 14-7 above. This resource is built automatically by the Panel Library Manager (PLM), after one has saved the drawing of a panel.

To vary the speed of the SMILEY ball, we will use the SUSPEND statement to suspend the task for varying fractions of a second. This is the SUSPEND_TIME that has been added to the bottom of the SMILE_DATA resource shown in Figure 14-8.

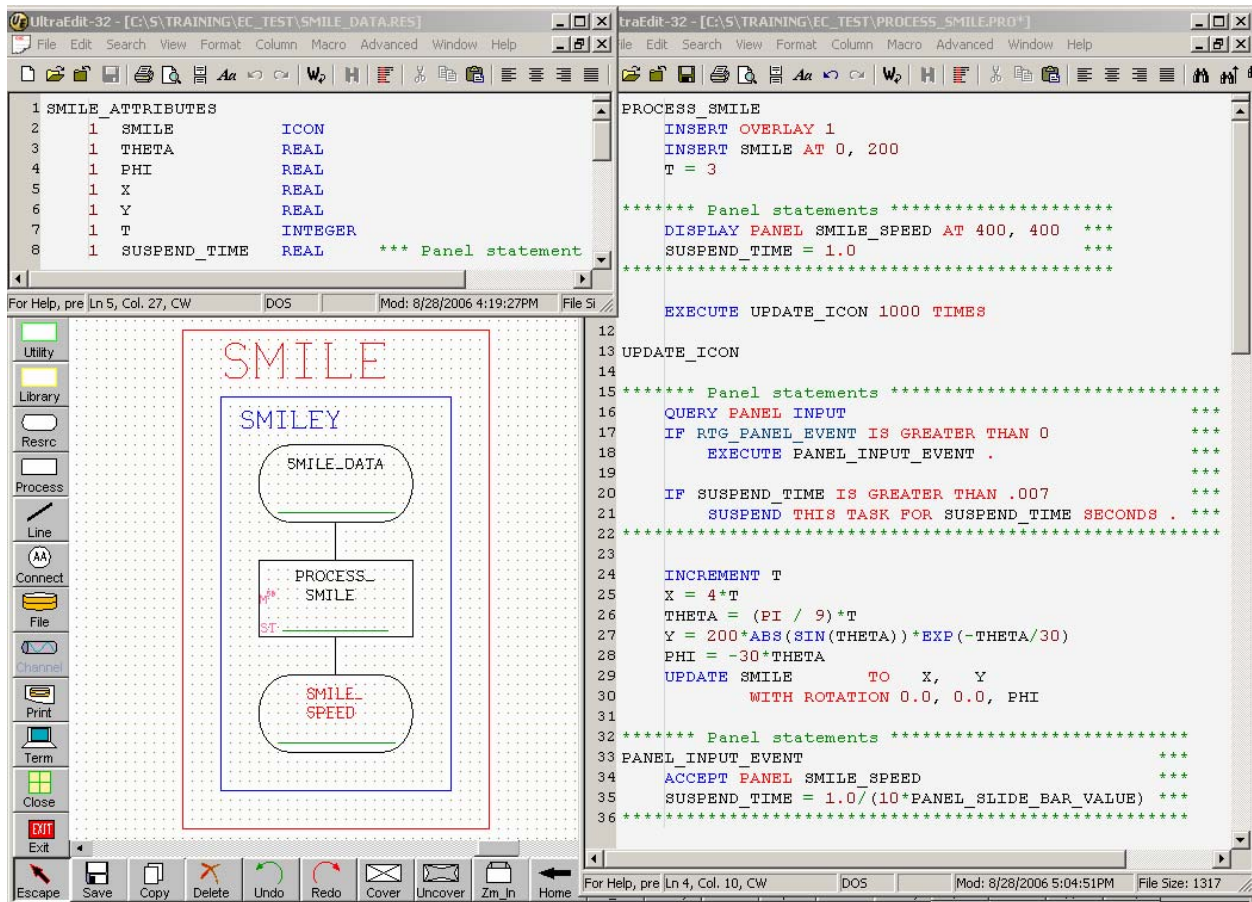


Figure 14-8. A new SMILE_SPEED resource for a slider bar to control SMILEY's speed.

Additional statements have been added to the process to display the panel, initialize the suspend time, query the panel for inputs, and update the suspend time if inputs have occurred.

The PLM is used to build the panel using a panel drawing board, shown in Figure 14-9, with the various widgets available to the user. The panel widgets are on the left column of buttons on the PLM drawing board. A vertical sliding bar has been used to adjust the speed. While the ball is bouncing, the user can adjust the speed of the ball by moving the slider up and down.

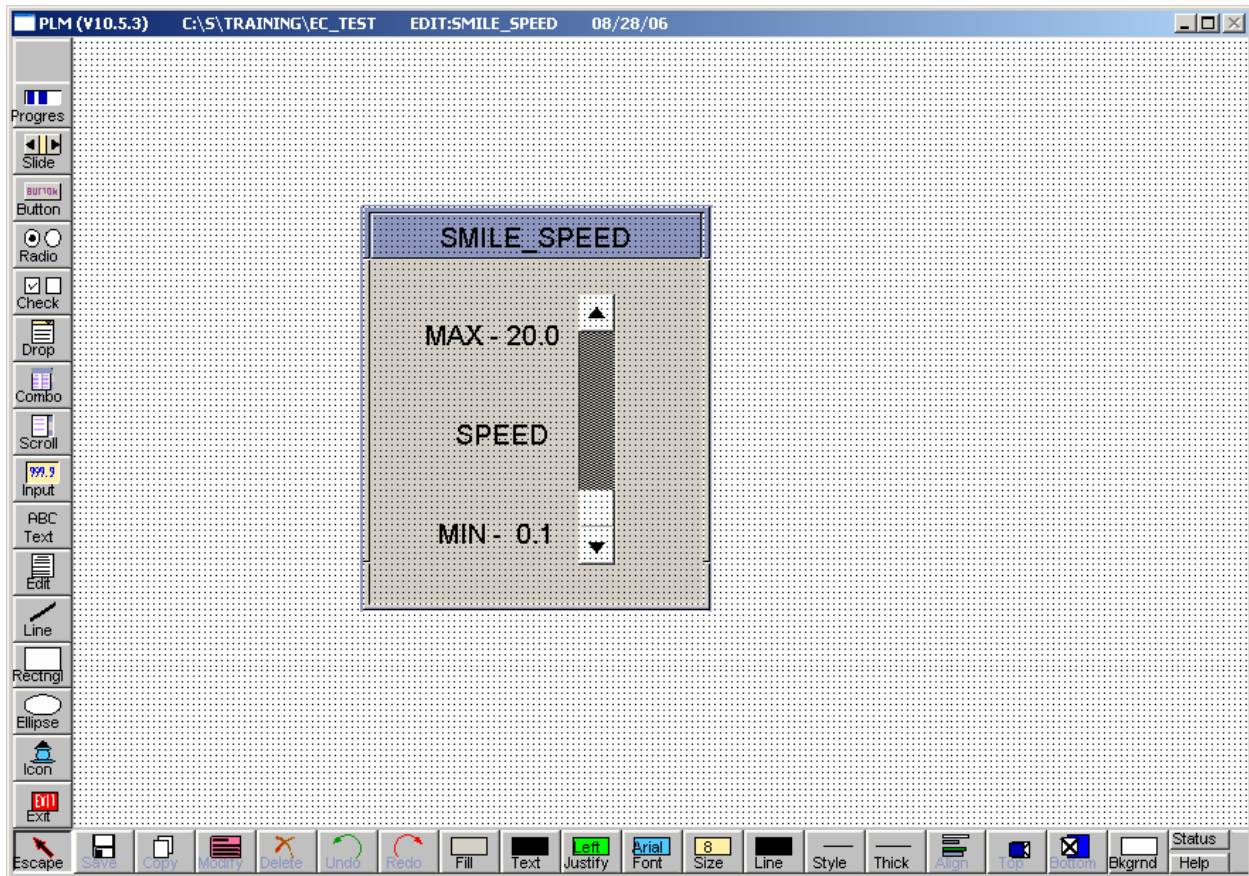


Figure 14-9. Building the slider bar panel in the PLM to control SMILEY's speed.

To build the slider bar, one just drags out the slider bar widget and places it in the panel, selecting the vertical option in this case. Prompted inputs allow selection of the minimum and maximum values of the bar. The user can add text to put the labels MAX - 20.0, SPEED, and MIN - 0.1. After saving the panel, the panel resource, SMILE_SPEED gets built with all of the statements needed to use the widgets in the panel. So all of the code that a user must write to build this example is shown above. That's it!

TIC TAC TOE GAME

In this example we will build an interactive game of Tic Tac Toe. The board will be built as a background overlay and the X and O letters will be built as icons that can be inserted on the board. This is shown in Figure 14-10 below. To determine where a player has placed the X or O icon, we must test the position of the icon relative to the board. Our test will simply be to determine if the center of the icon lies within one of the blue squares. If so, we will take it and center it, provided the grid square is empty.

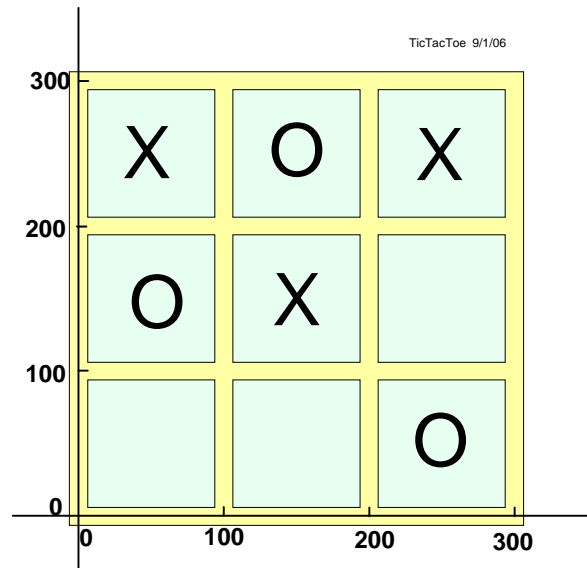


Figure 14-10. The game of TIC TAC TOE.

BUILDING THE GAME BOARD BACKGROUND OVERLAY

The (x, y) coordinate system for the game board is shown in Figure 14-10. It has been designed for ease of testing the placement of an icon. Given that the grid squares are each 84 units on a side, then all sides are 8 units away from the 0, 100, 200, 300 lines. Valid placement implies that the center of an icon must be in one of the ranges [8, 92], [108, 192], and [208, 292] in both the x and y directions. The background overlay is composed of 10 squares, the outer square and the 9 inner squares. This library module is rather simple to build. It is shown in Figure 14-11 below. The process is in the upper left corner and the resource in the lower right.

To start the game, each player will be given five icons of X or O. To place an icon on the board, the player left mouse clicks on one of the icons to select it, and holding the button down, drags it to place it over the desired grid square. When the button is released, the position of the icon is tested to ensure that it lies in one of the grid squares that is free. If not, it will not be accepted, and a beep will occur telling the player he has not selected a valid position. The player must then click on the icon to select it and drag it to a valid square, where it will be automatically centered.

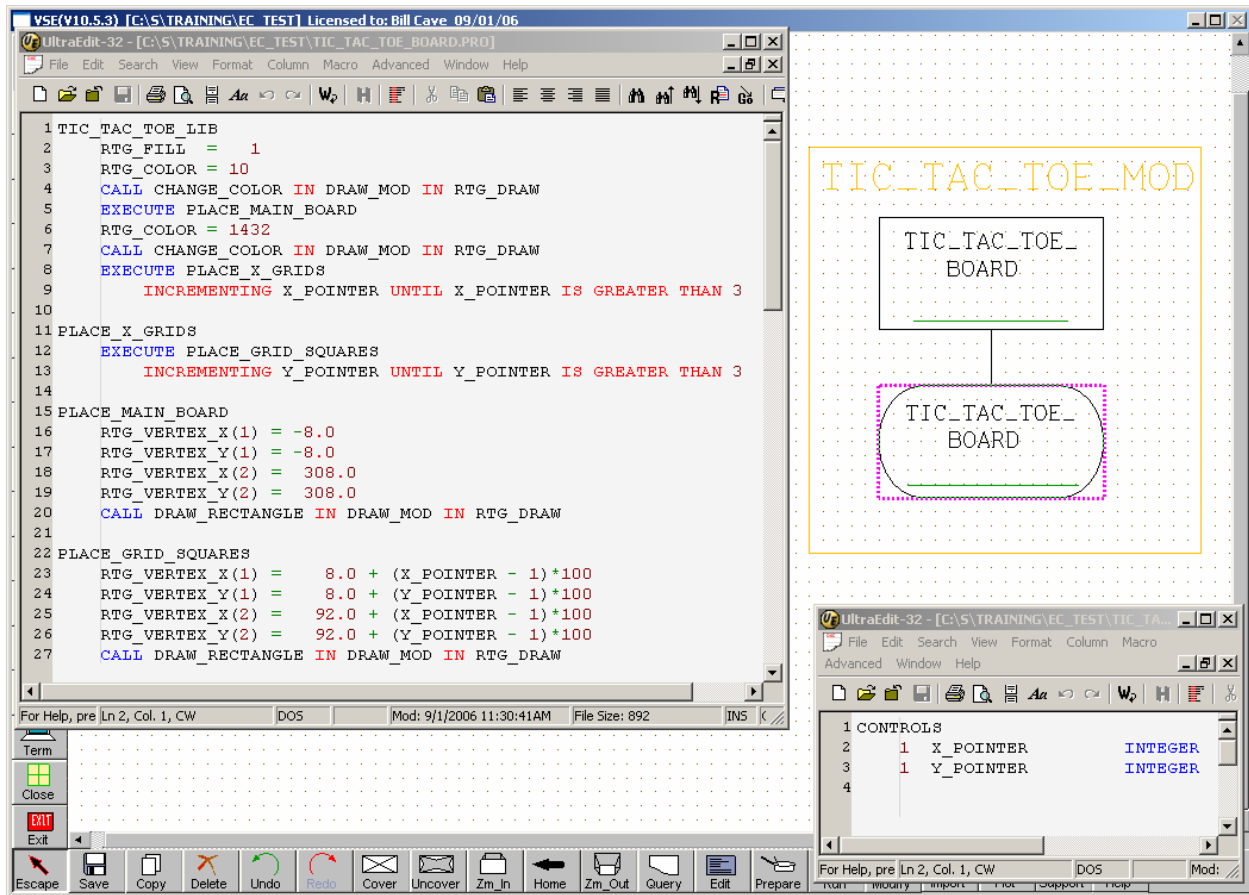


Figure 14-11. TIC TAC TOE background overlay module.

After the icon is centered, a test will be made to determine if the insertion has completed the game. If not, an icon from the other player's set must be selected and placed. The game continues until one of the players wins or it is determined that no one can win.

BUILDING THE GAME

The game starts with each player's icons next to the board as shown in Figure 14-12. As icons are placed, their positions are checked to ensure the center of the icon lies within an unoccupied blue box. Valid entries are centered automatically, and the state of the board is updated. A panel indicates the state of the game, and also contains a button that can end the game at any time.

After each valid entry, the state of the board is checked to determine if there is a winner. If so, a red line is drawn through the entries that won the game. This is illustrated in Figure 14-13. Otherwise, the game continues.

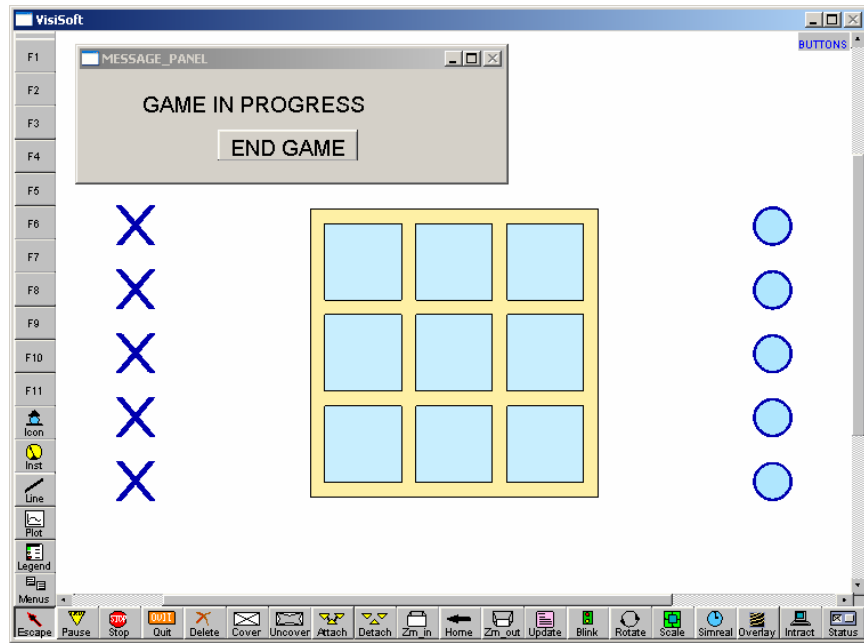


Figure 14-12. TIC TAC TOE starting screen.

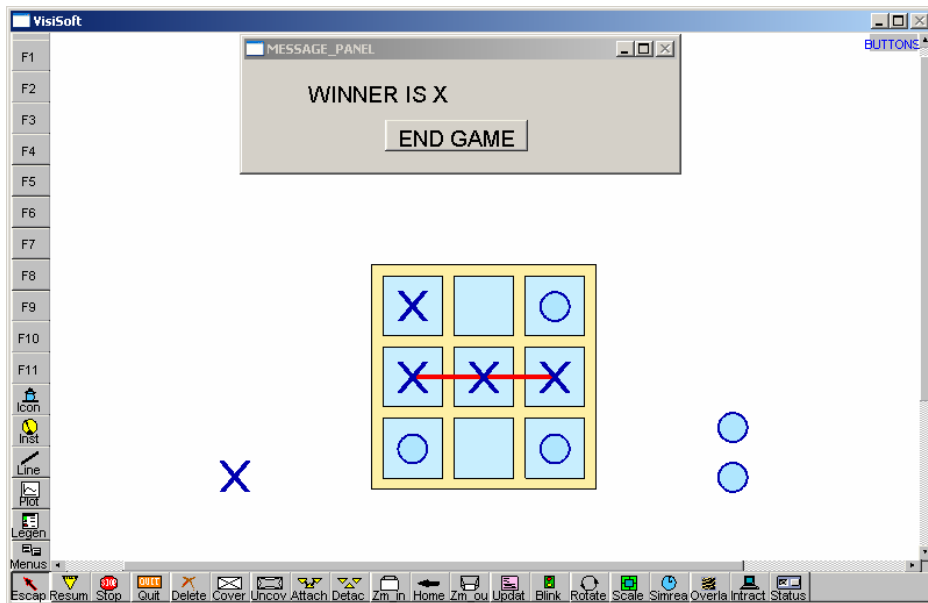


Figure 14-13. TIC TAC TOE ending screen.

The architecture is shown in Figure 14-14, with the two resources TIC_TAC_TOE and GAME_STATE. The MESSAGE_PANEL resource is not shown since it is built automatically using the PLM. Although it must be referred to by process TIC_TAC_TOE that controls the panel, the references are quite simple.

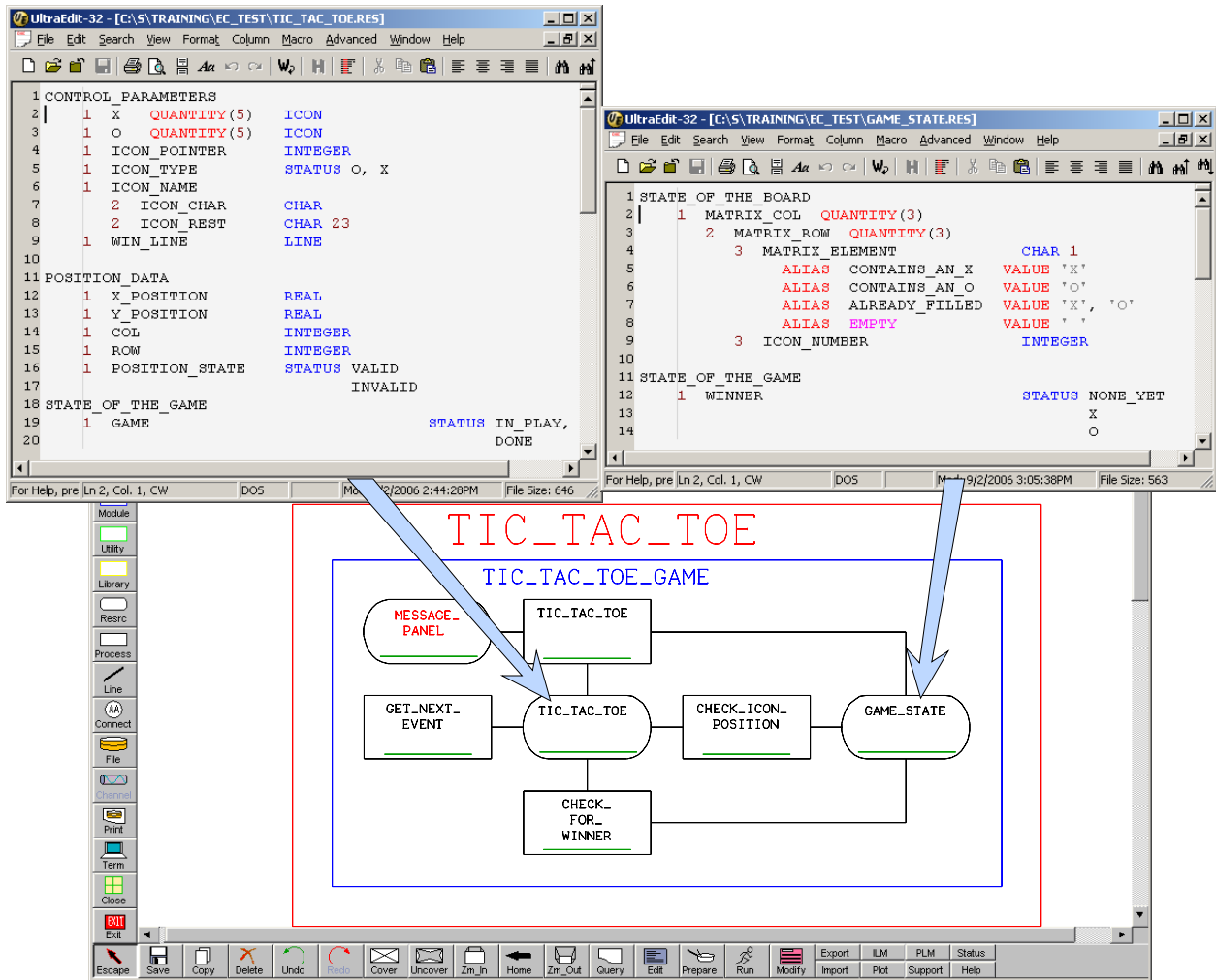


Figure 14-14. TIC TAC TOE ending screen.

The process TIC_TAC_TOE is shown in Figures 14-15a & b. It is the most complex of the processes. The rule GET_NEXT_EVENT starts with the statement

```
GET NEXT EVENT AND WAIT
```

This causes the task to wait on an RTG event. When an event occurs, the process specified as the RTG EVENT HANDLER (in this case: GET_NEXT_EVENT - Figure 14-16) is invoked, after which control is returned to the statement following the GET NEXT EVENT AND WAIT statement in TIC_TAC_TOE.


```

TIC_TAC_TOE
    EXECUTE INITIALIZE
    EXECUTE GET_NEXT_EVENT
        UNTIL GAME IS DONE

*****

INITIALIZE
    MOVE SPACES TO STATE_OF_THE_BOARD
    SET GAME      TO IN_PLAY
    SET WINNER    TO NONE_YET
    INSERT OVERLAY 1
    EXECUTE INSERT_X_O_ICONS
    MOVE 'GAME IN PROGRESS' TO MESSAGE_PANEL PANEL_TEXT
    DISPLAY PANEL MESSAGE_PANEL AT 800, 100

INSERT_X_O_ICONS
    EXECUTE INSERT_ICONS
        INCREMENTING ICON_POINTER
        UNTIL ICON_POINTER IS GREATER THAN 5

INSERT_ICONS
    X_POSITION = -200
    Y_POSITION = 360 - 70 * ICON_POINTER
    INSERT X(ICON_POINTER) ICON AT X_POSITION, Y_POSITION
    X_POSITION = 500
    INSERT O(ICON_POINTER) ICON AT X_POSITION, Y_POSITION

*****

GET_NEXT_EVENT
    GET NEXT EVENT AND WAIT
    QUERY PANEL INPUT
    IF RTG_PANEL_EVENT IS GREATER THAN 0
        EXECUTE PANEL_INPUT_EVENT .

    IF GAME IS DONE
        EXIT THIS RULE .

    CALL GET_NEXT_EVENT
    EXECUTE PROCESS_ICON_MOVE
    IF POSITION_STATE IS VALID
        CALL CHECK_FOR_WINNER .

    IF WINNER IS NONE_YET
        EXIT THIS RULE
    ELSE EXECUTE GAME_IS_OVER .

PROCESS_ICON_MOVE
    .
    .
    .

```

Figure 14-15a. Top part of process TIC_TAC_TOE.

```

      .
      .
      .
PROCESS_ICON_MOVE
  SET POSITION_STATE TO INVALID
  CALL CHECK_ICON_POSITION
  IF POSITION_STATE IS INVALID
    EXIT THIS RULE .

  IF ICON_TYPE IS X
    UPDATE X(ICON_POINTER) ICON
      TO X_POSITION, Y_POSITION
  ELSE
  IF ICON_TYPE IS O
    UPDATE O(ICON_POINTER) ICON
      TO X_POSITION, Y_POSITION .

*****

PANEL_INPUT_EVENT
  ACCEPT PANEL MESSAGE_PANEL
  IF MESSAGE_PANEL PANEL_BUTTON_STATUS IS ON
    SET GAME TO DONE .

GAME_IS_OVER
  IF WINNER IS X
    MOVE 'WINNER IS X' TO MESSAGE_PANEL PANEL_TEXT
  ELSE
  IF WINNER IS O
    MOVE 'WINNER IS O' TO MESSAGE_PANEL PANEL_TEXT
  ELSE
    MOVE 'ERROR'          TO MESSAGE_PANEL PANEL_TEXT .

  DISPLAY PANEL MESSAGE_PANEL
  SUSPEND THIS TASK FOR 4 SECONDS

```

Figure 14-15b. Bottom part of process TIC_TAC_TOE.

```

GET_NEXT_EVENT
  IF RTG_GRAPHICS_SYMBOL IS AN ICON
    EXECUTE GET_ICON_DATA .

GET_ICON_DATA
  MOVE RTG_ICON_SIMULATION_NAME TO ICON_NAME
  MOVE RTG_ICON_INSTANCE_PTR(1) TO ICON_POINTER
  IF ICON_CHAR IS EQUAL TO 'X'
    SET ICON_TYPE TO X
  ELSE IF ICON_CHAR IS EQUAL TO 'O'
    SET ICON_TYPE TO O .
  MOVE RTG_ICON_X TO X_POSITION
  MOVE RTG_ICON_Y TO Y_POSITION

```

Figure 14-16. Process GET_NEXT_EVENT.

The two processes, CHECK_ICON_POSITION and CHECK_FOR_A_WINNER are relatively simple to build. CHECK_ICON_POSITION checks to see if the center of the icon is within one of the boxes and if so, checks if there is already an icon in that grid square. If it is inside and the square is free, it is placed in the center. If not, it is put back where it was before it was selected. If placed in a grid square, it then calls CHECK_FOR_A_WINNER to determine if that icon has caused a win. This is done by checking the possible win combinations to determine if one is a win.

CONVERSION TO A NETWORKED GAME

The game described above provides a single mouse input and window output, awkward for two players. This can be converted to a two-platform game using a VisiSoft Interprocessor Resource. The architecture is shown in Figure 14-17. Each player has a corresponding task that provides the state of the game in the window, and takes control from and passes control to the other player. So the architecture shown below is repeated on the O side. This runs in a client-server mode, where the server is started before the client. This concept is easily adapted to multi-player games.

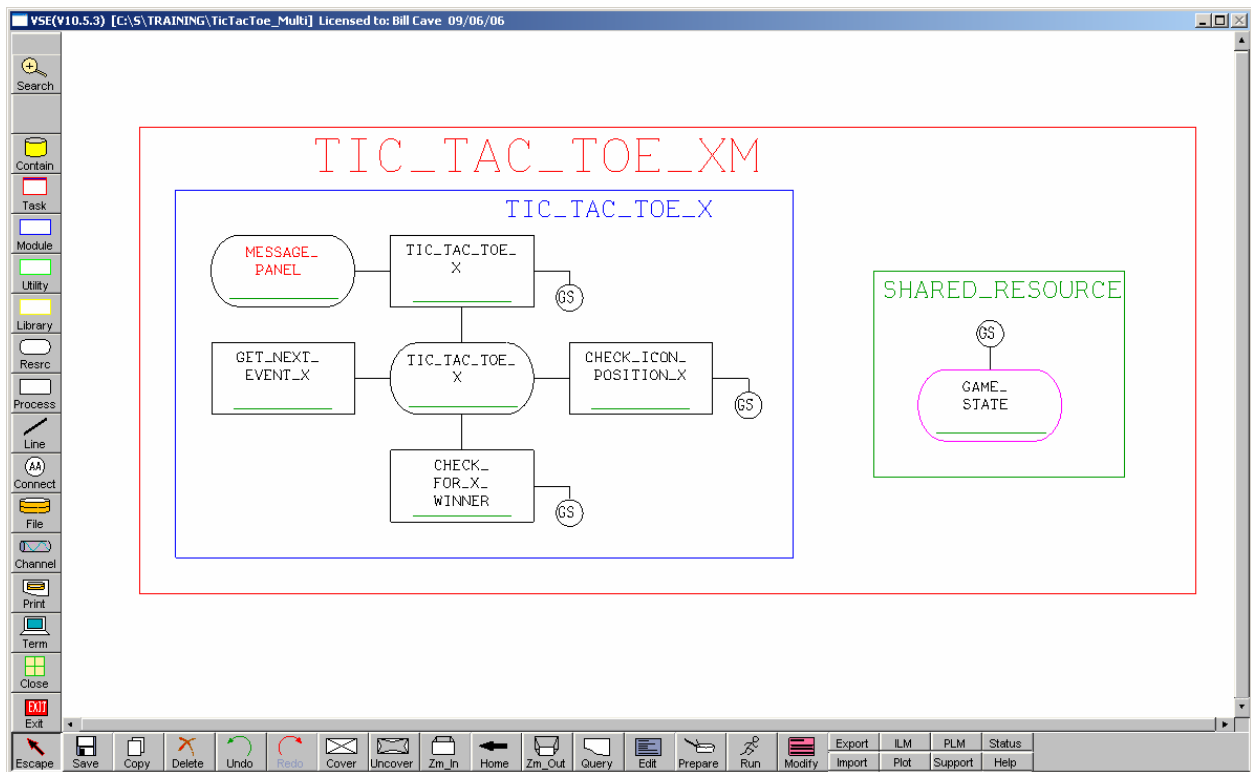


Figure 14-17. Multi-platform architecture.

THE ICON LIBRARY MANAGER (ILM)

The examples of icons built using the ILM above are somewhat trivial. It is worth while to consider some others. Examples of more complex icons are shown in Figure 14-18 below.

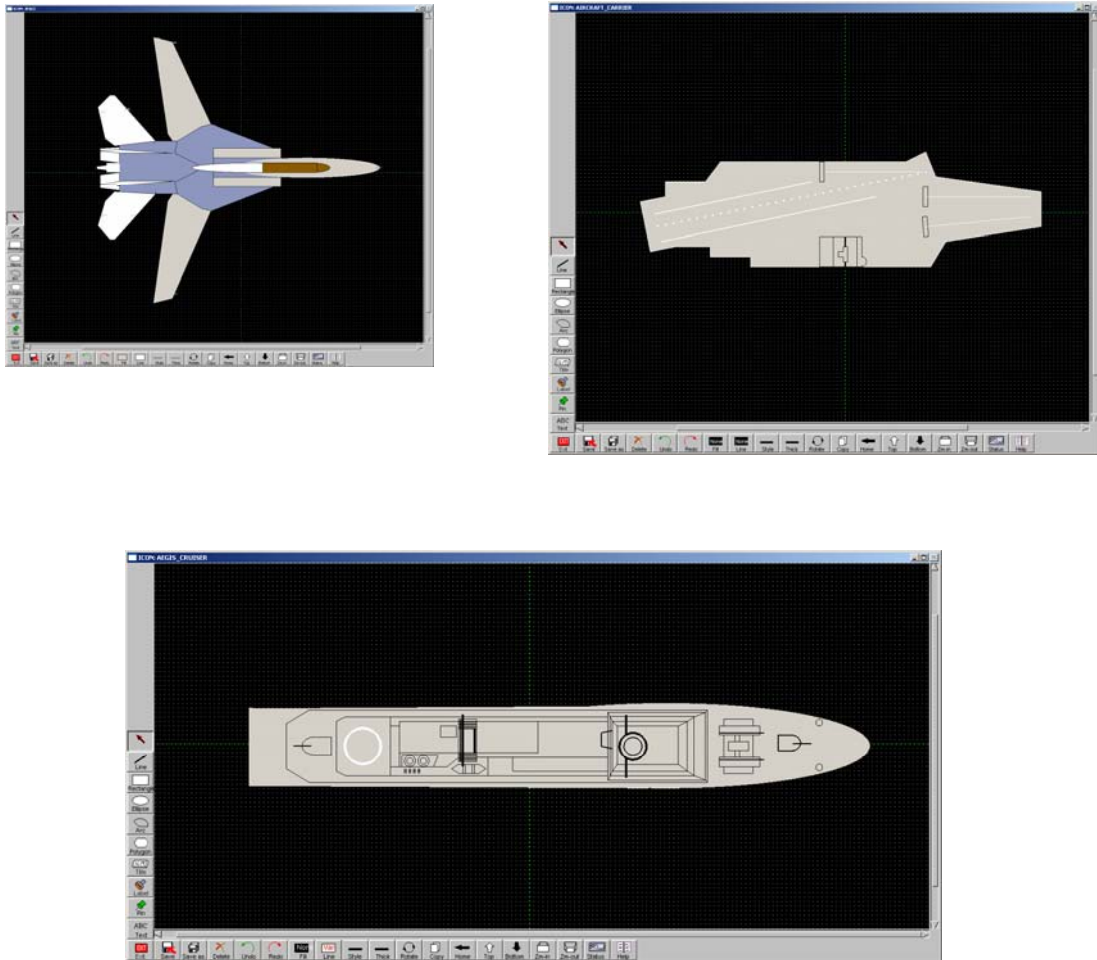
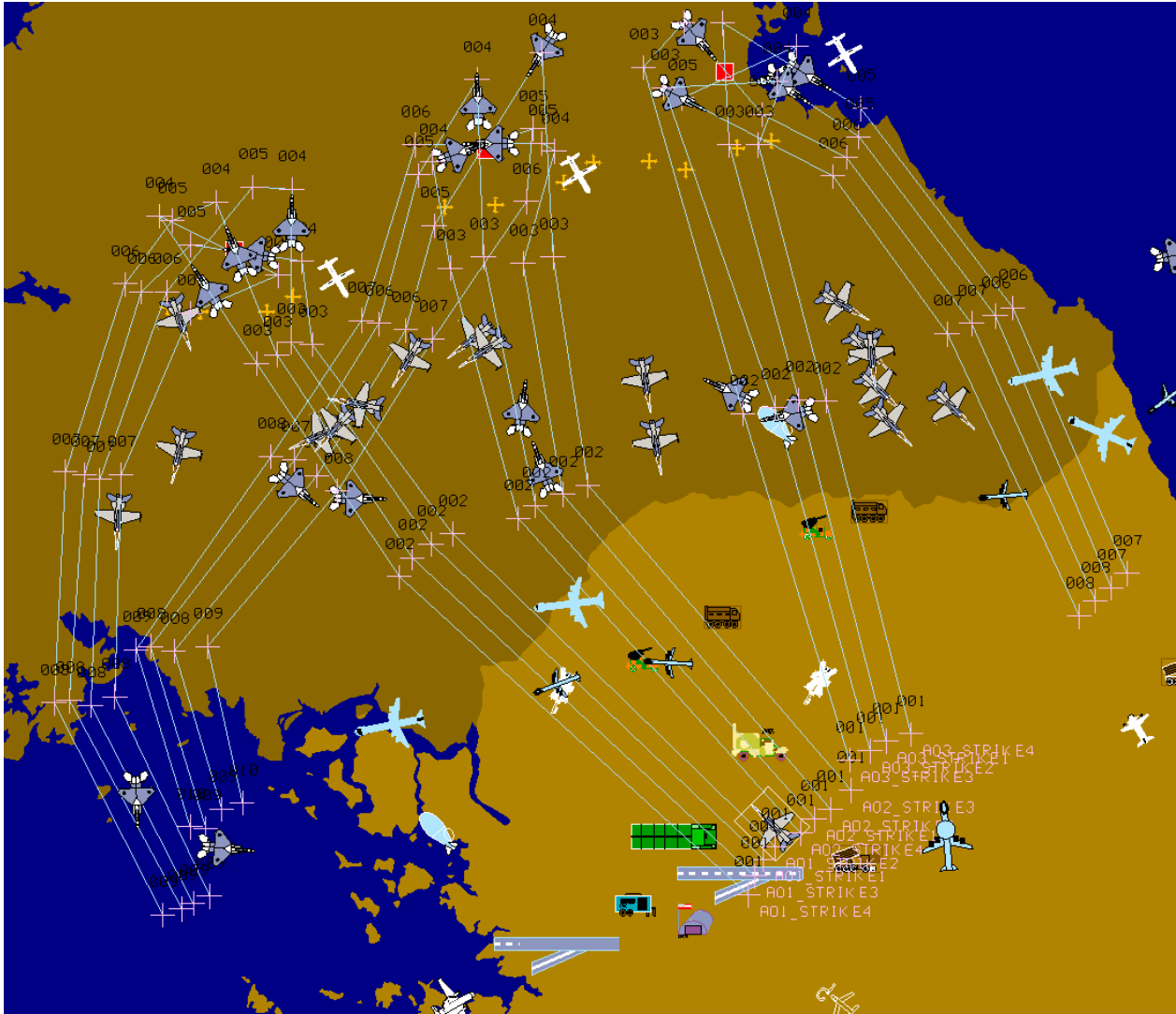


Figure 14-18. Examples of more complex icons built using the ILM.



Chapter 15. Simulation

This chapter describes the use of the General Simulation System (GSS) in building discrete event simulations. Although GSS supports the discrete event approach, it provides for dynamic nonlinear mathematical modeling within a simulation. This includes the solution of stiff nonlinear systems of differential equations with “look ahead” algorithms. However, most of today’s system designs require that sophisticated sets of IF ... THEN ... ELSE ... rules are built into the algorithms, such as those used to provide layered protocols. This is particularly true in the design of complex communications and control systems.

In this chapter we start with a well known example typically solved using differential equations. We then extend the concepts used to in this example to the simplified design and test of embedded algorithms used in a telephone network.

RABBIT - COYOTE BIOLOGICAL MODEL COMPARISON

In this example, we compare a continuous-time system model to its rule-based counterpart. We will use the classical example of biological balance between a host and a parasite as provided in many texts, e.g., Gordon, [43], pp. 103. In this example, the dynamics of the interactivity between the rabbit and coyote populations are modeled. In this model, rabbits are the *hosts* (prey), multiplying in large numbers compared to coyotes that are the *parasites* (hunters). The equations, when simplified, take the form described by Gordon as follows.

$$\frac{dr}{dt} = A \cdot r(t) - B \cdot r(t) \cdot c(t)$$

$$\frac{dc}{dt} = K \cdot r(t) \cdot c(t) - D \cdot c(t)$$

The first equation defines the rate of change of the rabbit population, where rabbit births are a fraction, A, of the existing population, and rabbit deaths (due to coyote kills) are a fraction, B, of the product of the rabbit and coyote populations. The coyote population changes similarly, but they are modeled as the birth rate being a fraction, K, of the product of the rabbit and coyote populations, and their death rate is a fraction, D, of their population.

Figure 15-1 illustrates an approach to describing the model graphically using a fairly standard *analog diagram* for the differential equations. This set of equations can be solved using special methods or existing software systems. The analog diagram is easily related to the equations.

Figure 15-2 shows a *stock and flow diagram* for the same system, using slightly different coefficients for the equations. This diagram is somewhat more easily related to the stock and flow of rabbits and coyotes, but is harder to relate to the system of equations.

The classical approach to determining the coefficients for this problem is to assume the solution to be quasi-stable, i.e., oscillatory, with no damping to a stable state. This is justified on the grounds that oscillation is observed in real life. However, as we shall show, this is not a realistic representation of the physical system, since any perturbation will drive the system into an unstable state, causing at least one of the populations to go to infinity or zero. In fact, basically stable systems may appear to operate in constant oscillation, even though they require continuous perturbation from an external source. One merely has to redefine the external source as part of the overall system. Any form of clock or electronic oscillator is good example. This is fine when using simple mathematical models of oscillators as examples in a classroom environment, where the complexity of nonlinear models need not be described. However, it presents a misleading picture when trying to explain the real biological behavior of interest here. And this is another case where

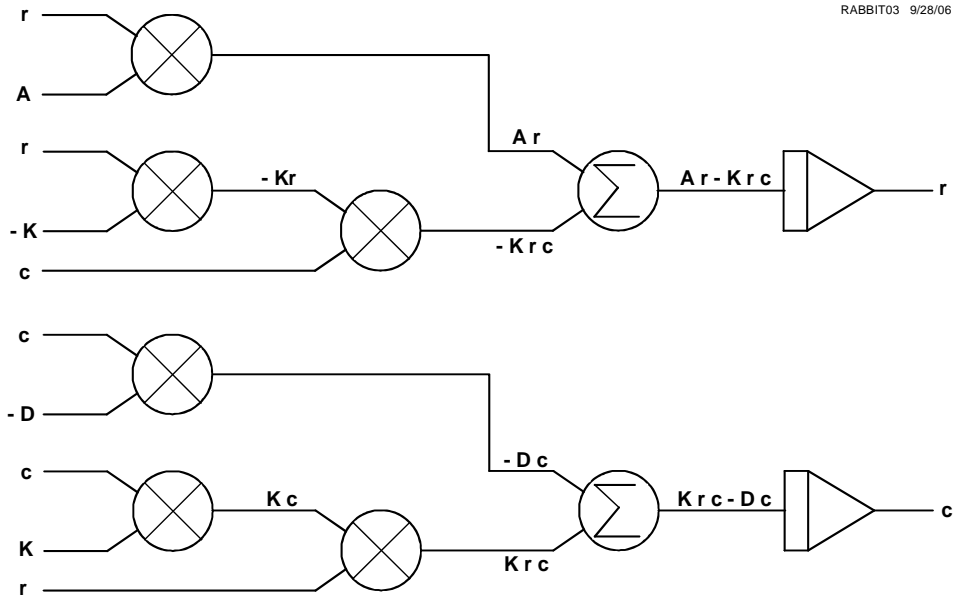


Figure 15-1. Rabbit - coyote biological model using analog symbol diagram.

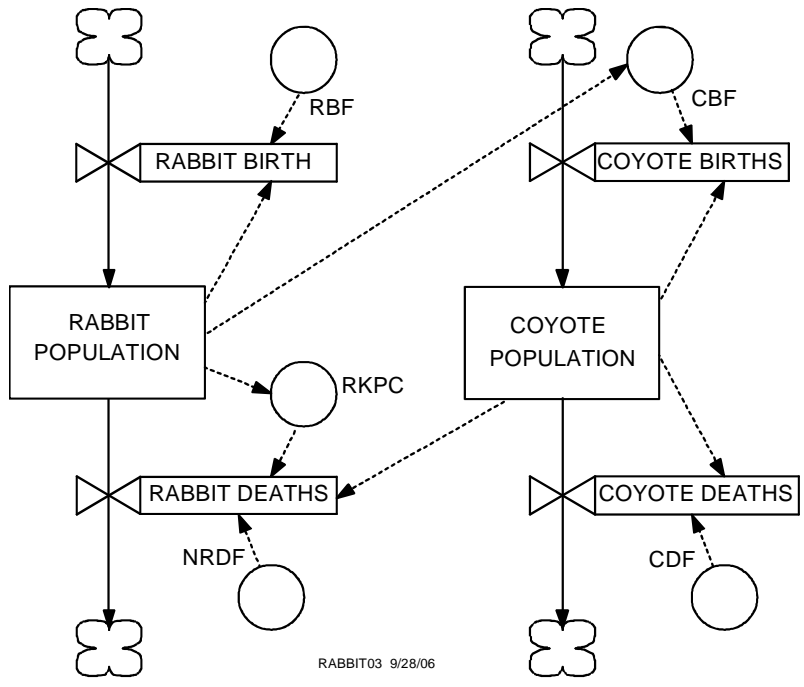


Figure 15-2. Rabbit - coyote biological model using system dynamics symbol diagram.

A MORE REALISTIC MODEL OF THE PHYSICAL PHENOMENON

The theory and design of electronic oscillators has been well researched. Their operational characteristics are governed by nonlinear physical phenomenon, refer to Hafner, [49]. Accurate representation of real physical oscillatory behavior requires nonlinear models. In addition, damping exists in all physical systems to some degree, as do external perturbations. When the perturbations are absent, the system will relax, with decreasing oscillatory behavior, to a stable state - normally not oscillatory because of the effects of damping. When a perturbation occurs, the system moves from its stable state into what may appear to be oscillatory motion that naturally decays. These perturbations can come close enough together to cause superposition of their effects, and give the appearance of continual oscillatory motion. We submit that is the case with the typical biological model.

The model of a system that is less than critically damped will show the same form of oscillatory responses every time it is perturbed. Clearly, biological systems such as rabbits and coyotes are always being perturbed by external factors not modeled here. These can produce what would appear to be continuous oscillation, even though the systems themselves are highly stable. These additional perturbations would hardly change the overall behavior of the system. Depending on how one chooses the coefficients in the nonlinear equations, vastly different results can occur. One must study the effects of perturbations on the populations to gain good agreement with reality.

Given these facts, both of the linear models described above can misrepresent the real physical behavior of the biological system, particularly if one is concerned about studying the survival of the populations. It is more realistic to represent coyote deaths as due to starvation (not enough rabbit kills) as well as natural causes. Similarly, given reasonable circumstances, the rabbit population in a linear model quickly grows to infinity - an impossible consequence if we are studying a finite geographical area, with a finite food supply.

Rabbits will also starve if they don't have enough food, and can die of natural causes as well. Also, the incubation periods of different animals can be significantly different, affecting the time constants for birth after pregnancy. As we add these more detailed representations to gain accuracy, the model will necessarily become more complex. But, instead of solving a set of equations for fictitious coefficients to land on the single point of oscillation, we are providing real characterizations, observed phenomena and watching the simulated results. Furthermore, both of the prior approaches require a knowledge of how to transform the description of a physical problem into differential equation format, and then find a means to solve it. Our rule-based approach dispenses with this requirement. The description appears in a natural language format, requiring only a knowledge of algebra.

A RULE-BASED RABBIT-COYOTE BIOLOGICAL MODEL

Figure 15-3 below shows a GSS version of the rabbit-coyote simulation using the same biological type model. This approach is totally different than that using differential equations described in the prior section. Note that we are modeling the physical phenomena directly, minimizing abstractions. This is a great aid when it comes to adding detail to the model. As a result, the processes in the model relate directly to pregnancy, birth, natural death, and of course the rabbit hunt, as they affect the population of the herds.

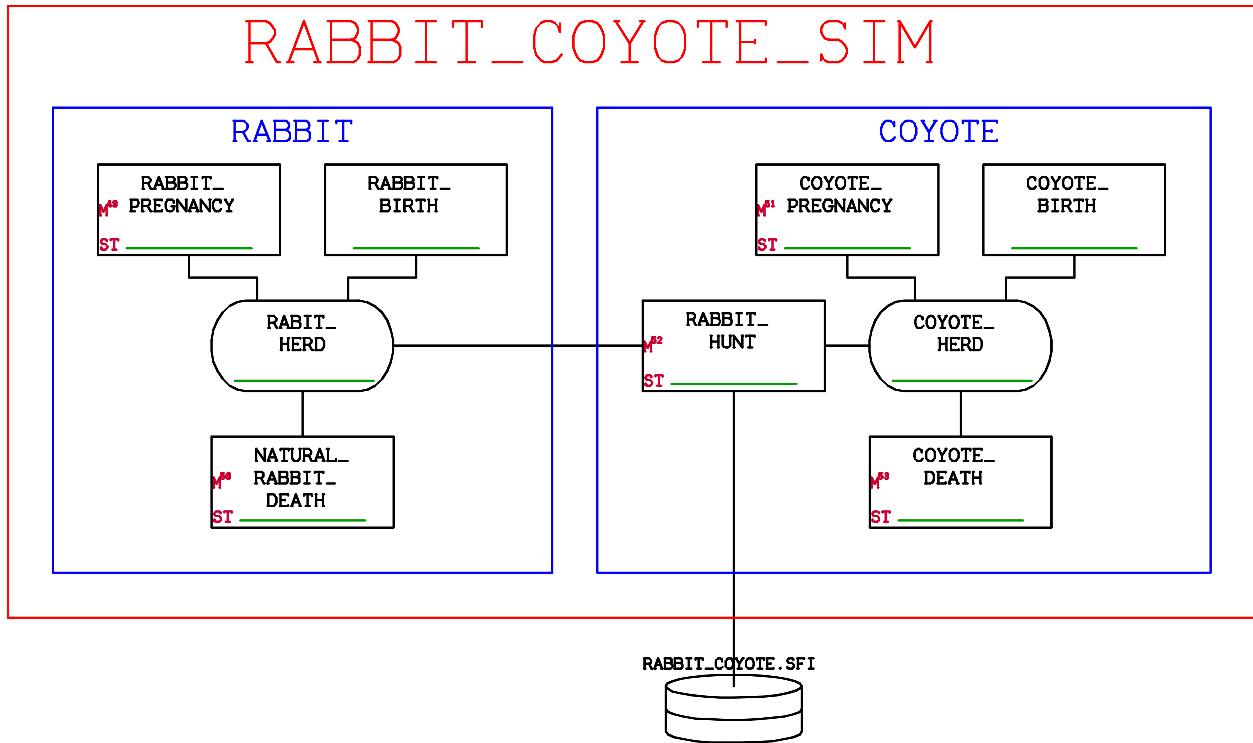


Figure 15-3. Rabbit - coyote biological model using the GSS Model Development Graphics.

The GSS Processes rules and Resource data structures are shown on the next three pages. Although not many decision processes are represented in these simple models, the approach to the computations is more understandable in terms of real life considerations. One does not have to understand differential equations to represent the physical system. However, the models account for many more details than their counterparts using an abstract mathematical approach. Equally important, the counterpart of nonlinear differential equations would be much more complex to write and solve.

We also note that using discrete event simulation, most of these processes run independently, transforming the states of the herds when they run. The pregnancy process affects the percent of pregnant animals in the herds. The birth process causes an increase in the herd size. The natural death process causes a reduction in the herd size, and the rabbit hunt causes a reduction in the rabbit herd. Note that the red codes inside processes cause them to be started since they run independently, scheduling themselves in the future.

Graphs of the dynamic behavior of the possible rabbit - coyote relationship, as represented in GSS, are shown in Figures 15.4 through 15-7. These results were obtained by modifying the birth, death, and hunger submodels for each. It can be seen that a wide range of results can be obtained, depending upon various factors. Figures 15-4 and 15-5 show the result of a single perturbation, at the beginning of the chart, on this very stable system. We note that a sequence of perturbations, *occurring two years apart*, will give the appearance of continuous oscillation. We would expect perturbations to occur at least this often.

```
RESOURCE: RABBIT_HERD
```

```
RABBIT
```

```
1 POPULATION INTEGER INITIAL_VALUE 10000
1 PREGNANCY_SET INDEX INITIAL_VALUE 1
1 PREGNANCIES QUANTITY(9) INTEGER
1 NATURAL_DEATHS INTEGER
1 TOTAL_DEATHS INTEGER
1 HUNGER_DEATHS INTEGER
1 HUNGER_DEATH_FACTOR REAL
1 DEATHS_BY_COYOTE INTEGER
1 REPRODUCTION_RATE REAL INITIAL_VALUE 0.2
1 NATURAL_DEATH_RATE REAL INITIAL_VALUE 0.05
1 MEDIAN_POPULATION INTEGER INITIAL_VALUE 10000
```

```
PROCESS: RABBIT_PREGNANCY
```

```
PREGNANCY_CONTROL
```

```
*** DETERMINE PREGNANCY GROUP (MONTH - INSTANCE)
    IF PREGNANCY_SET IS GREATER THAN 2
        PREGNANCY_SET = 1.
*** COMPUTE NUMBER OF PREGNANCIES FOR THIS PERIOD
    PREGNANCIES (PREGNANCY_SET) = REPRODUCTION_RATE * POPULATION

*** SCHEDULE BIRTH AND PREGNANCY DATES
    SCHEDULE RABBIT_BIRTH IN 60 DAYS USING PREGNANCY_SET
    SCHEDULE RABBIT_PREGNANCY IN 30 DAYS USING PREGNANCY_SET
    INCREMENT PREGNANCY_SET.
```

```
PROCESS: RABBIT_BIRTH
```

```
RABBIT_BIRTH_CONTROL
```

```
ADD PREGNANCIES(PREGNANCY_SET) TO POPULATION
```

```
PROCESS: NATURAL_RABBIT_DEATH
```

```
RABBIT_DEATH_CONTROL
```

```
IF POPULATION IS GREATER THAN ZERO
```

```
    HUNGER_DEATH_FACTOR = ((MEDIAN_POPULATION + POPULATION)/  
                           MEDIAN_POPULATION)**2
```

```
    TOTAL_DEATHS = NATURAL_DEATH_RATE * POPULATION *  
                  HUNGER_DEATH_FACTOR
```

```
    SUBTRACT TOTAL_DEATHS FROM POPULATION.
```

```
IF POPULATION IS LESS THAN 2
```

```
    STOP.
```

```
SCHEDULE NATURAL_RABBIT_DEATH IN 30 DAYS
```

```
RESOURCE: COYOTE_HERD
```

```
COYOTE
```

```
1  POPULATION                INTEGER INITIAL_VALUE 250  
1  PREGNANCY_SET             INDEX  INITIAL_VALUE  1  
1  PREGNANCIES QUANTITY(9)   INTEGER *** ALLOW UP TO 9 INSTANCES  
1  NATURAL_DEATHS           INTEGER  
1  HUNGER_DEATHS            INTEGER  
1  TOTAL_DEATHS             INTEGER  
1  RABBIT_KILLS             INTEGER  
1  HUNGER                   REAL  
1  COYOTE_RABBIT_RATIO      REAL  
1  REPRODUCTION_RATE        REAL  INITIAL_VALUE 0.1  
1  NATURAL_DEATH_RATE       REAL  INITIAL_VALUE 0.02  
1  HUNGER_DEATH_RATE        REAL  INITIAL_VALUE 0.02  
1  APPETITE                 INTEGER INITIAL_VALUE 20  
1  PROBABILITY_OF_CATCH     REAL  
1  MEDIAN_RABBIT_CATCH      INTEGER INITIAL_VALUE 10000
```

```
TIME_FACTORS
```

```
1  DAY_COUNT                INTEGER  
1  MONTH                    INTEGER
```

```
PROCESS: COYOTE_PREGNANCY
```

```
PREGNANCY_CONTROL
```

```
*** DETERMINE_PREGNANCY SET (MONTH - INSTANCES)
```

```
IF PREGNANCY_SET IS GREATER THAN 3  
    PREGNANCY_SET = 1.
```

```
*** SET COYOTE PREGNANCIES AND SCHEDULE BIRTH
```

```
PREGNANCIES(PREGNANCY_SET) = REPRODUCTION_RATE * POPULATION  
SCHEDULE COYOTE_BIRTH IN 90 DAYS USING PREGNANCY_SET  
SCHEDULE COYOTE_PREGNANCY IN 30 DAYS USING PREGNANCY_SET  
INCREMENT PREGNANCY_SET.
```

```
PROCESS: COYOTE_BIRTH

COYOTE_BIRTH_CONTROL
  ADD PREGNANCIES(PREGNANCY_SET) TO POPULATION
```

```
PROCESS: COYOTE_DEATH

COYOTE_DEATH_CONTROL
  IF POPULATION IS GREATER THAN ZERO EXECUTE
    COMPUTE_COYOTE_DEATHS.
  IF POPULATION IS LESS THAN 2
    STOP.
  SCHEDULE COYOTE_DEATH IN 30 DAYS

COMPUTE_COYOTE_DEATHS
  *** DETERMINE COYOTE HUNGER
  COYOTE_HUNGER = 3*(5*COYOTE_POPULATION/RABBIT_KILLS) ** 3
  NATURAL_DEATHS = NATURAL_DEATH_RATE * POPULATION
  HUNGER_DEATHS = COYOTE_HUNGER * HUNGER_DEATH_RATE * POPULATION
  TOTAL_DEATHS = NATURAL_DEATHS + HUNGER_DEATHS
```

```
PROCESS: RABBIT_HUNT

RABBIT_HUNT
  *** DETERMINE RABBIT_KILLS
  PROBABILITY_OF_CATCH = (RABBIT_POPULATION /
    (RABBIT_POPULATION + MEDIAN_RABBIT_CATCH)) ** 2
  RABBIT_KILLS = APPETITE * COYOTE_POPULATION * PROBABILITY_OF_CATCH
  DECREMENT RABBIT_POPULATION BY RABBIT_KILLS
  IF RABBIT_POPULATION IS LESS THAN 2
    STOP.
  IF RABBIT_KILLS ARE LESS THAN ZERO
    RABBIT_KILLS = 1.
  SCHEDULE RABBIT_HUNT IN 30 DAYS
```

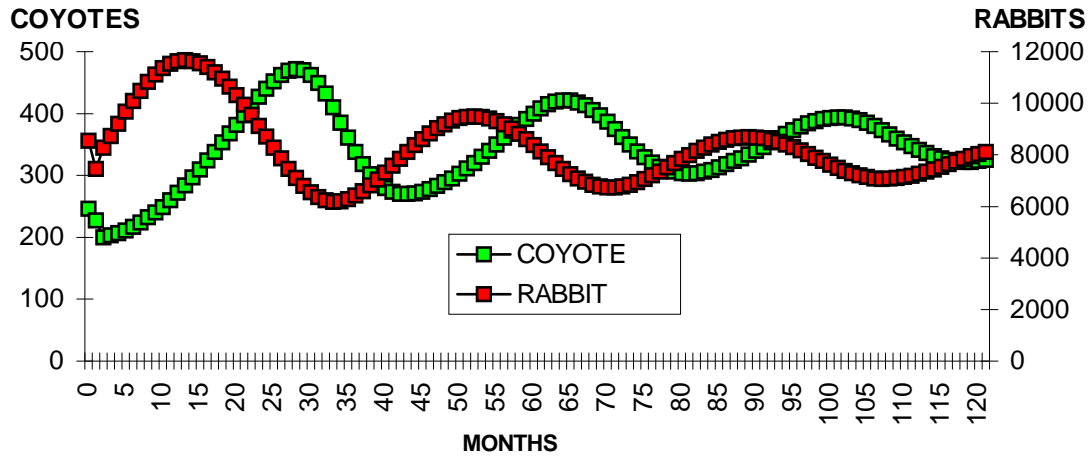


Figure 15-4 Oscillatory relationship when basic model is linear and rabbits are limited by food.

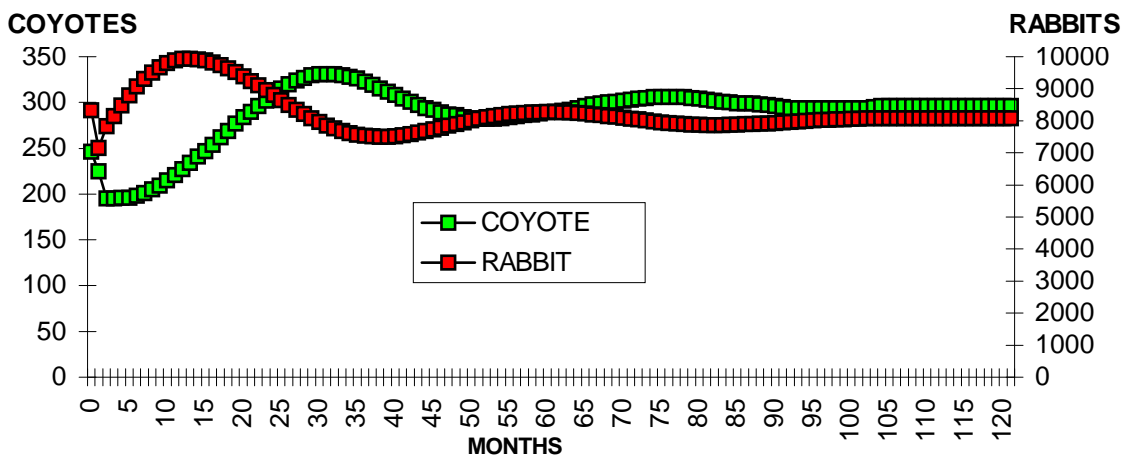


Figure 15-5 Stable relationship when basic model is nonlinear and rabbits are limited by food.

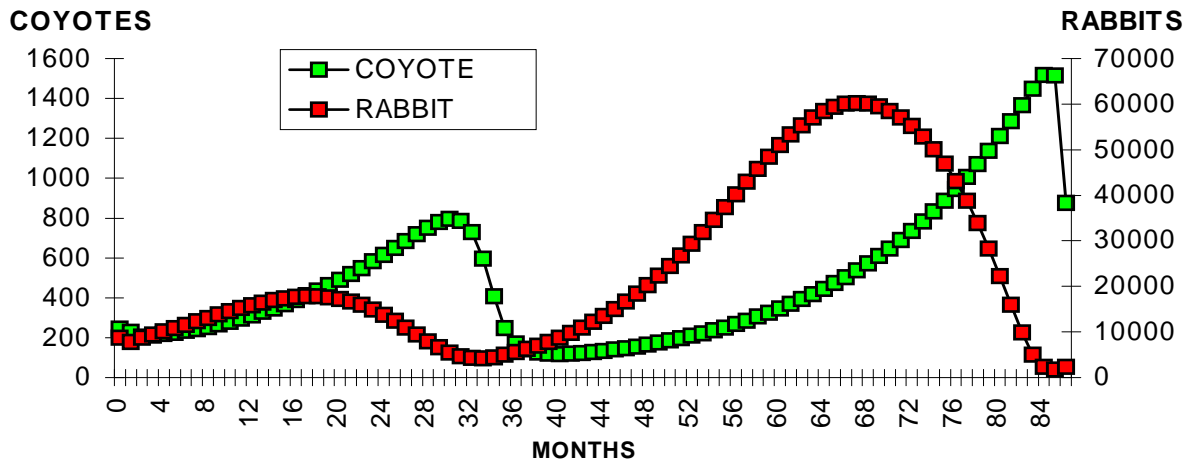


Figure 15-6. Unstable relationship when model is linear and rabbit growth is not limited.

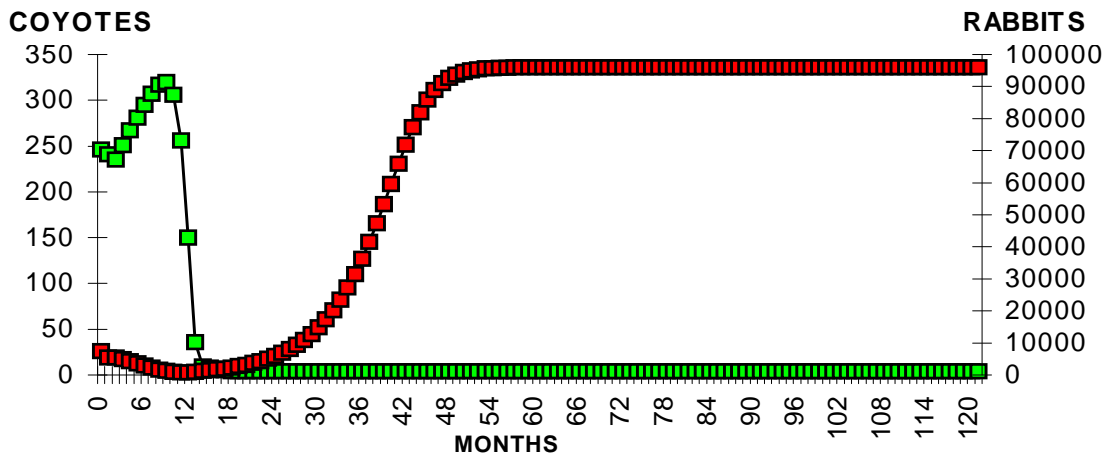


Figure 15-7. Extinction occurs when model is linear and rabbit growth is slow and limited.

TELEPHONE NETWORK SIMULATION

Figure 15-8 is the graphical output representing the telephone network simulation. This is a simulation of four business offices (green boxes), each with different numbers of people having phones. We are concerned with interoffice calls that must go through the PBXs (gray boxes) and the local telephone company's central switch (brown box). We must determine how many lines to buy between each office PBX and the central switch. In the picture, green lines are in use, black indicates call setup in progress, and tan represents unused lines.

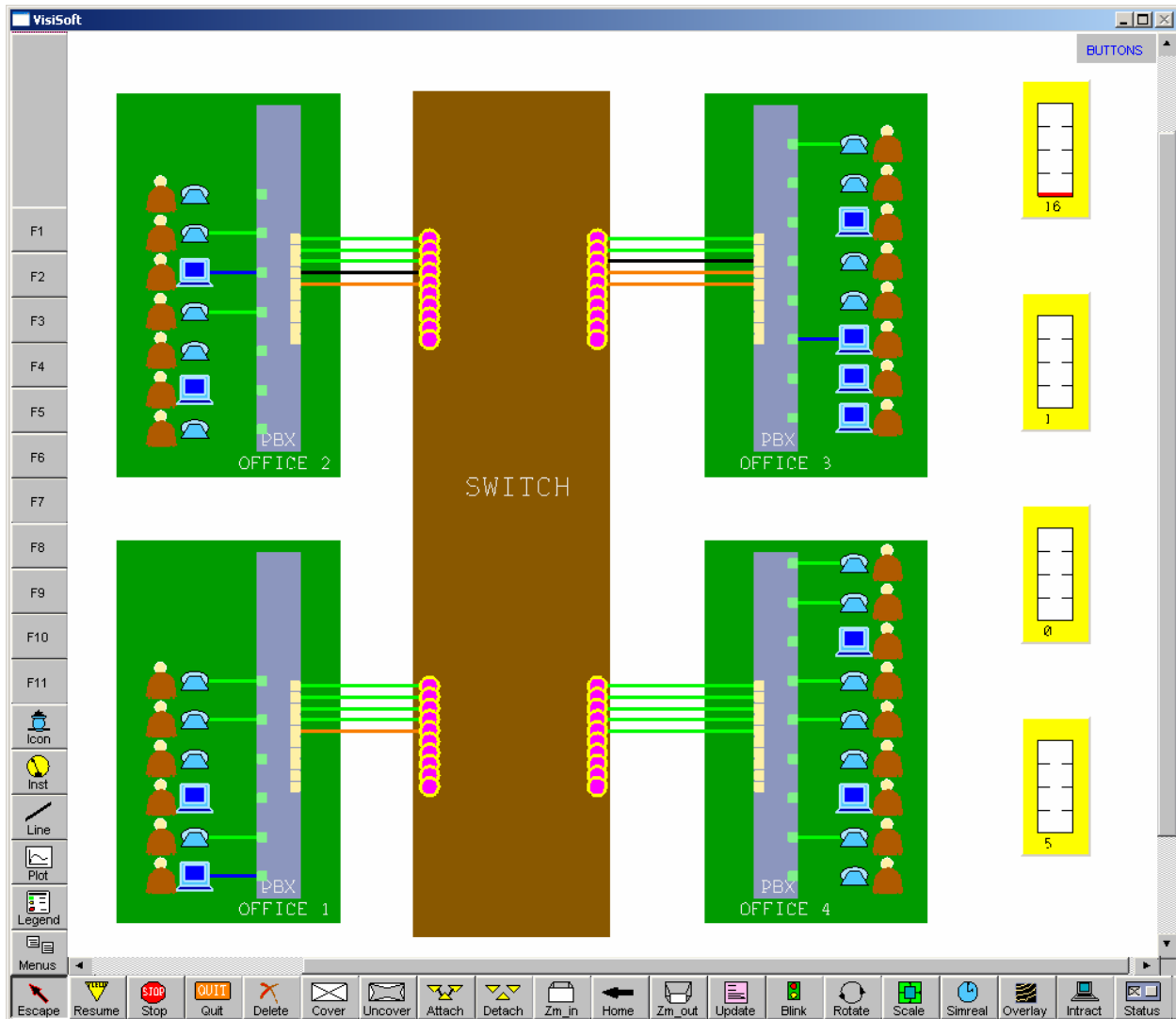


Figure 15-8. Graphical output representing the telephone network simulation.

To minimize our cost, we want to buy as few lines as needed to ensure a specified probability that calls will be completed under stress conditions (the busy hour). To do this, we must measure calls attempted, calls completed, calls that were blocked (not enough lines) and calls not completed because the other party was busy (not blocked).

The architecture for this simulation is shown in Figure 15-9. The simulation was designed to support up to ten local offices with up to twenty subscribers each. Each office has a PBX that is connected to the central switch. In addition, modules exist for initialization, running the busy hour scenario, providing dynamic graphical output, and collecting the data needed to determine the results.

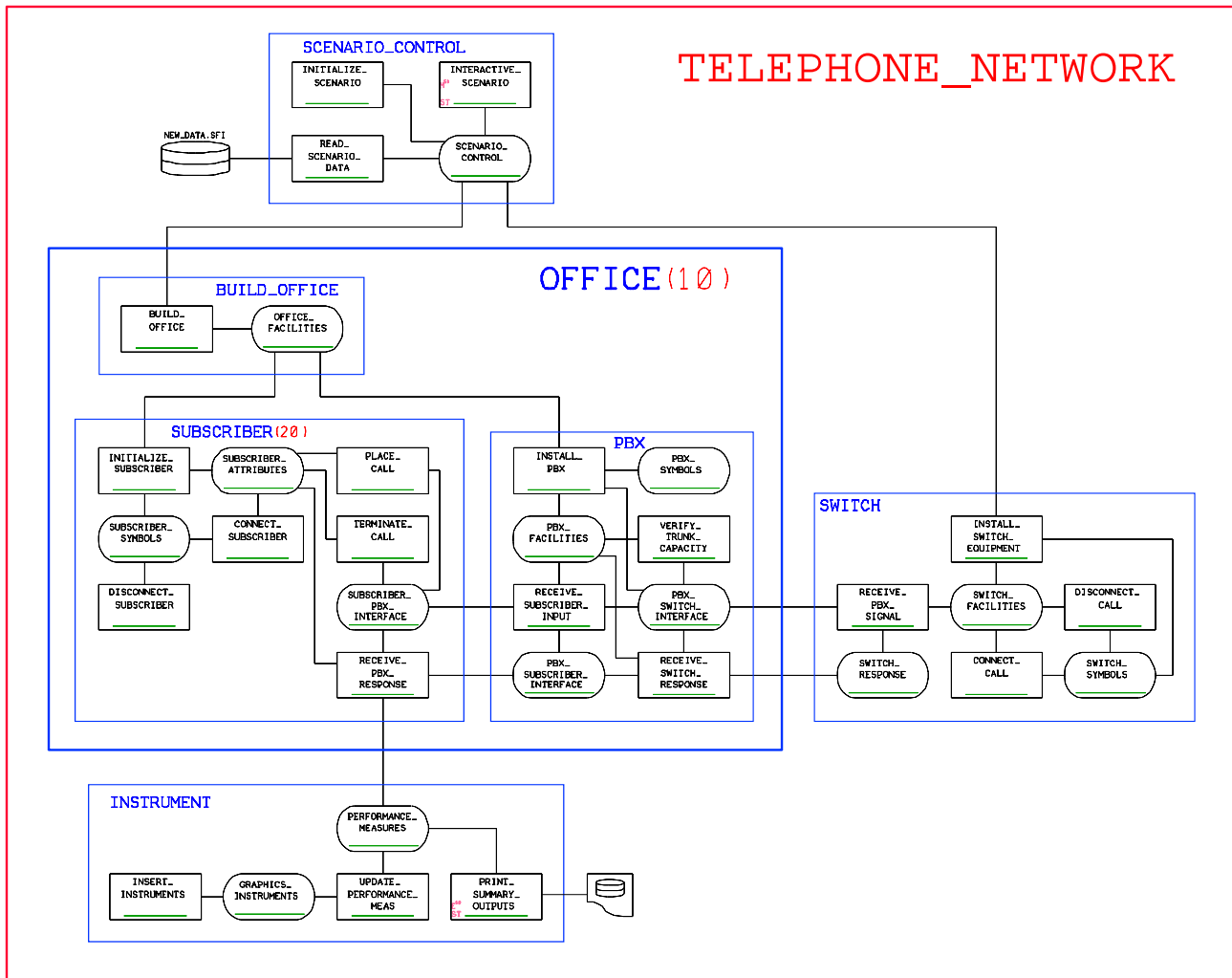


Figure 15-9. Architecture for the telephone network simulation.

A single simulation is used for a given design, i.e., a selected number of trunk lines between the PBX and central switch for each office. Then using the GSS optimization facilities, the number of trunk lines can be varied while running a set of simulations, e.g., 50 to determine the optimal solution. Then, having set the optimal solution, a Monte Carlo run can be performed using 50 simulations to create a distribution of outcomes to verify that the constraint on the desired probability is met.

VALIDITY OF THE MODELS

The most important factor in using simulation to perform design and testing is to ensure that simulated results match what happens under live (real) test conditions, i.e., that the simulation results are a valid representation of the system. As illustrated in Figure 15-10, simulation validity is determined by the validity of the measures of merit (MOM) obtained from the simulation. These measures, of equipment performance or overall system effectiveness, must quantify values that an analyst must use to make decisions. In order for these measures to be valid, they must be based on data from the simulation that is accurate relative to predicting what will occur in a real environment with real systems. This implies that the models and scenarios used in a simulation must provide a sufficiently accurate representation of the real system and its environment.

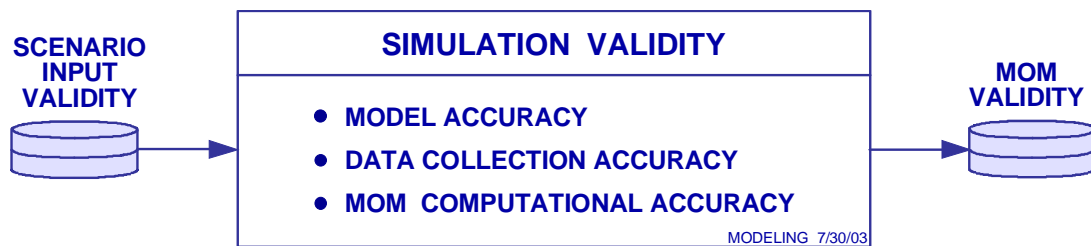


Figure 15-10. Determining Model Validity

Generally speaking, the major factors boil down to two: validity of the scenario, and accuracy of the models. With a good simulation design, many scenarios can be created by the users to investigate different worst cases. Typically, the critical piece is accuracy of the model representations. As one learns more about the problem, one must be able to increase model detail to account for more factors. The ease with which this is accomplished depends directly upon the simulation environment.

REUSABILITY OF THE MODELS

More importantly, one would like to pull a model off the shelf and reuse it in different simulations. The ability to do this depends directly upon the range of validity of the model. This implies that models with sufficient detail can be used for a wider array of problems than those tailored simply to a specific application. An example in communications is a propagation model that covers a very broad band of the frequency spectrum. Such a model is typically much more complex than one that covers a small band of the spectrum. But to be reusable, models must be understandable and independent. These properties are defined below.

MODEL INDEPENDENCE

Looking at the architecture in Figure 15-9, one observes two factors. First, the models are built along the physical lines of the system being modeled. The office model contains the model for subscribers and PBXs. This model is connected to the switch model. The instrument model is connected to the subscriber model, since that is where the calls are initiated and the resulting blocked, busy, and connected signals are recorded. The scenario control module is connected to the office and switch. The number of connections between models never exceeds two, making it relatively easy to disconnect a model and reconnect it in another simulation.

MODEL UNDERSTANDABILITY

This property determines the ability of an analyst or modeler, other than the original author, to understand the model to the extent that it is easily validated and reused. From an economic standpoint, models that are more easily understood are more valuable because they are more easily validated, modified, and reused.

We note that the most complex model from an element standpoint is the subscriber model. Within that model, placing calls is the most complex. Let's look at the SUBSCRIBER_ATTRIBUTES resource and the PLACE_CALL process in Figures 15-11 and 12, noting that one of the resources that PLACE_CALL uses is not shown.

SUBSCRIBER	INDEX		
SUBSCRIBER_INFORMATION			
1	CALLERS_PLAN	STATUS	PLACE_NEW_CALL RETRY_CALL
1	SUBSCRIBER_TYPE	STATUS	DATA VOICE
1	SUBSCRIBER_STATUS	STATUS	BUSY FREE
CURRENT_CALL_PARAMETERS			
1	CALL_TIME	REAL	
1	CALL_START_TIME	REAL	
1	CALL_DURATION	REAL	
1	PHONE_NUMBER	STATUS	UNKNOWN FOUND
CALL_ATTRIBUTES			
1	CALL_INTERGEN_TIME	REAL	***INITIAL_VALUE 12 ***MINUTES
1	AVERAGE_CALL_DURATION	REAL	***INITIAL_VALUE 4 ***MINUTES
1	VARIANCE	REAL	***INITIAL_VALUE 1 ***MINUTE
1	RETRY_INTERGEN_TIME	REAL	***INITIAL_VALUE 4 ***MINUTES
PHONE_BOOK			
1	PHONE_BOOK_STATE	STATUS	INCOMPLETE COMPLETE
1	PHONE_TOTAL_OFFICES	INDEX	
1	PHONES_IN_OFFICE	QUANTITY(4) INDEX	
DIALED_OFFICE			
		INDEX	
DIALED_SUBSCRIBER			
		INDEX	

Figure 15-11. Resource: SUBSCRIBER_ATTRIBUTES.

```

PLACE_CALL
  IF SUBSCRIBER_STATUS IS FREE
    EXECUTE MAKE_CALL
  ELSE EXECUTE RETRY_LATER .

MAKE_CALL
  IF CALLERS_PLAN IS PLACE_NEW_CALL
    SET PHONE_NUMBER TO UNKNOWN
    EXECUTE LOOK_UP_NUMBER UNTIL PHONE_NUMBER IS FOUND .
  MOVE OFFICE TO SUBSCBR_PBX_SRCE_OFFICE
  MOVE SUBSCRIBER TO SUBSCBR_PBX_SRCE_SUBSCBR
  MOVE DIALED_OFFICE TO SUB_PBX_DES_OFF
  MOVE DIALED_SUBSCRIBER TO SUBSCBR_PBX_DEST_OFFICE
  SET SUBSCRIBER_STATUS TO BUSY
  SET SUBSCRIBER_SIGNAL TO PLACE_CALL
  MOVE CLOCK_TIME TO CALL_START_TIME
  SCHEDULE RECEIVE_SUBSCRIBER_INPUT NOW
  CALL CONNECT_SUBSCRIBER

RETRY_LATER
  SCHEDULE PLACE_CALL IN EXPON(RETRY_INTERGEN_TIME) SECONDS

LOOK_UP_NUMBER
  DIALED_OFFICE = (PHONE_TOTAL_OFFICES * RANDOM) + 1
  IF DIALED_OFFICE IS EQUAL TO OFFICE
    EXIT THIS RULE .
  DIALED_SUBSCRIBER =
    (PHONES_IN_OFFICE(DIALED_OFFICE) * RANDOM) + 1
  SET PHONE_NUMBER TO FOUND

```

Figure 15-12. Process: PLACE_CALL.

The SUBSCRIBER_ATTRIBUTES resource contains most of the important information on the state of a subscriber. The two other resources within the SUBSCRIBER model are SUBSCRIBER_SYMBOLS, which contains the subscriber icon information, and SUBSCRIBER_PBX_INTERFACE, which contains state information to be sent to the PBX. We note that these attributes have been selected to help make the PLACE_CALL process easily understood by a third party. The most difficult part is the transfer of information from the subscriber to the PBX regarding the calling subscriber's own number and office number, as well as the numbers of the called office and subscriber. We note that routing tables are not needed to obtain the desired measures from this simulation.

To provide for additional insight into these models, the process RECEIVE_SUBSCRIBER_INPUT is also provided in Figure 15-13. The intent here is to demonstrate the readability of the rules. Although this simulation has been characterized as simple, the algorithms for passing information through the system, so that all of the control messages necessary to set up a call are modeled, are not so simple. However, any engineer who has an understanding of these algorithms can quickly learn the logic of the models, and determine their validity or reusability.

```

RECEIVE_SUBSCRIBER_INPUT
  IF SUBSCRIBER_SIGNAL IS PLACE_CALL          *** SOURCE
    EXECUTE ATTEMPT_CONNECTION
  ELSE IF SUBSCRIBER_SIGNAL IS END_CALL       *** DESTINATION
    EXECUTE BREAK_CONNECTION.

ATTEMPT_CONNECTION
  IF TRUNKS_AVAILABLE ARE GREATER THAN 0
    EXECUTE ESTABLISH_CONNECTION
  ELSE EXECUTE CONNECTION_FAILURE.

ESTABLISH_CONNECTION
  SET PBX_SUBSCRIBER_LINE TO BUSY
  DECREMENT TRUNKS_AVAILABLE
  MOVE SUBSCRIBER_MESSAGE TO PBX_SWITCH_MESSAGE
  SET PBX_SWITCH_SIGNAL TO PLACE_CALL
  SCHEDULE RECEIVE_PBX_SIGNAL NOW

CONNECTION_FAILURE
  SET PBX_SUBSCRIBER_SIGNAL TO BLOCKED_AT_SOURCE
  SCHEDULE RECEIVE_PBX_RESPONSE NOW
  USING SUB_PBX_SRC_SUB

BREAK_CONNECTION
  INCREMENT TRUNKS_AVAILABLE
  MOVE SUBSCRIBER_MESSAGE TO PBX_SWITCH_MESSAGE
  SET PBX_SWITCH_SIGNAL TO END_CALL
  SET PBX_SUBSCRIBER_LINE TO FREE
  SCHEDULE RECEIVE_PBX_SIGNAL NOW

```

Figure 15-13. Process: RECEIVE_SUBSCRIBER_INPUT.

We should also point out that all of the statements are intuitive except for the SCHEDULE statement. This statement schedules a process to be run at a future time (n seconds from now) or at the current time (NOW). This causes the process name to be placed in a queue at a specified time and priority (not used above). Also stored in the queue are up to six instance pointers so that when the process runs, it knows what model instances it represents. These instance pointers are determined in the architecture (VDE) environment. The selection of the instance pointers is shown in Figure 15-14, where the names SOURCE and DESTINATION are selected as the first and second instance pointers.

Thus, when one process schedules another, the values of the instance pointers are automatically passed from one to the other through the scheduler. The modeler only has to use a common name for these pointers and set them to the desired value where necessary. This makes the models much easier to write as well as read.

To make all this work, behind the scenes all such resources are instanced, i.e., there is a copy for each instance. So every model instance has a separate copy of all the resources contained within it. If a model is not instanced, then there is only one copy. But if there are up to 10 subscribers each in 4 offices, there are 40 instances of subscriber resources. This makes running simulations very efficiently on parallel processors also very simple. In fact, the modeler need not even think about that problem. This is discussed in Chapter 16.

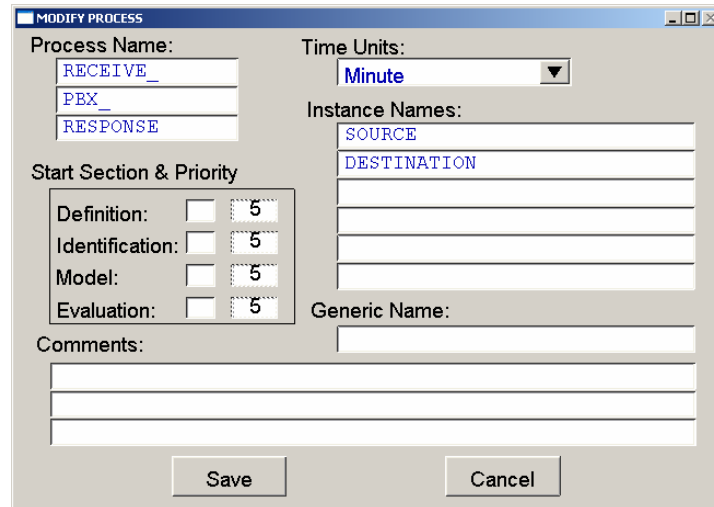


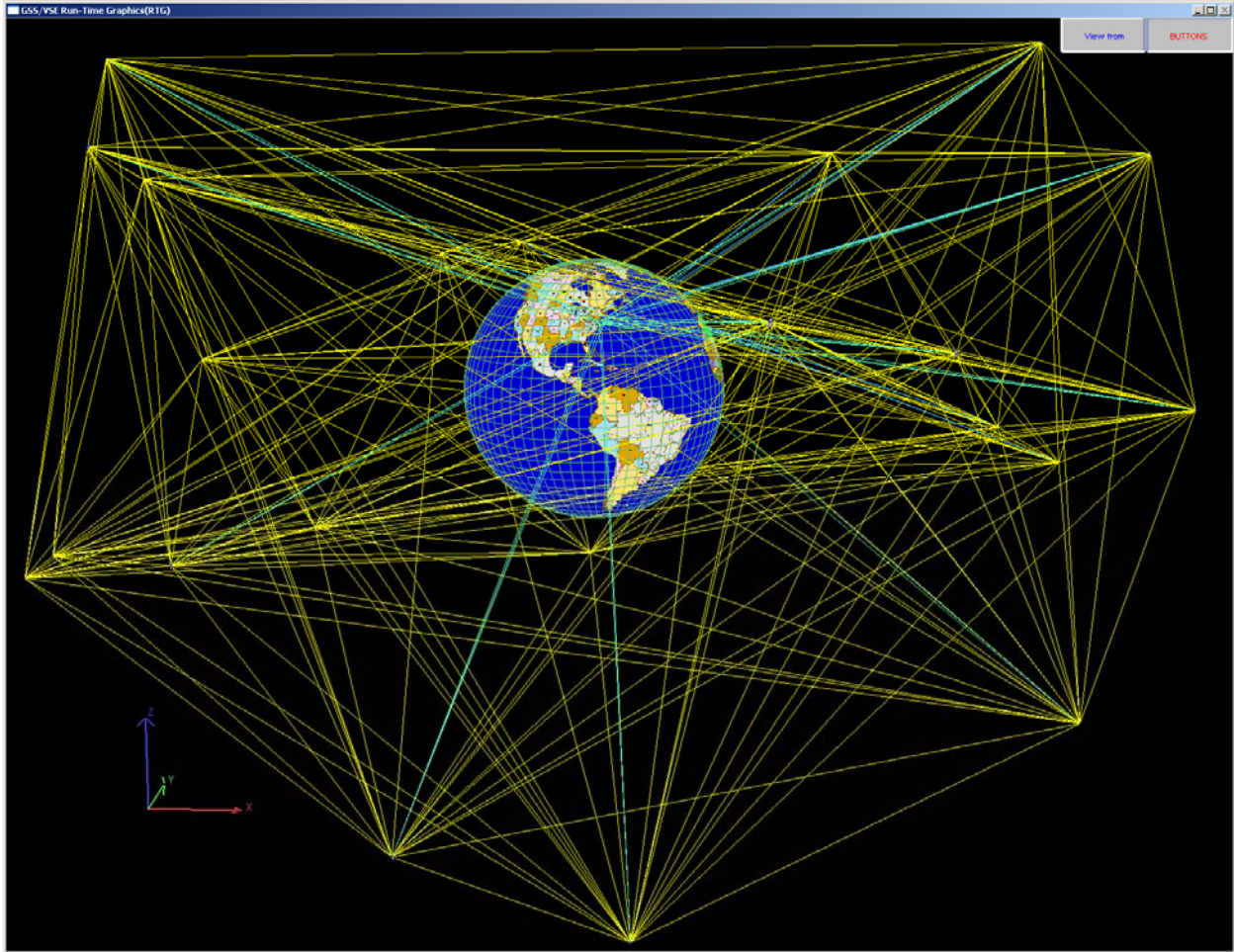
Figure 15-14. Selection of Instance Names for RECEIVE_PBX_RESPONSE.

INTERACTING WITH THE SIMULATION

In this particular simulation, the user can interact with the simulation while it is running. This can be done in a few ways. One can click on the ICON button (in the left set of buttons in Figure 15-1). This brings up a list of icons that are available to the simulation. If a user selects a subscriber icon, it can be inserted into any of the offices provided there is room for more subscribers (not all are active). When it is clicked down, a subscriber initialization process takes place to activate it as part of the running simulation. It can then initiate and receive calls. Thus, an analyst can watch what happens when more subscribers are added to an office. Similarly, one can add or subtract trunk lines interactively and watch the results. The ability to make changes interactively aids in the analysis of complex systems.

DESIGNING AND TESTING COMPLEX SYSTEM ALGORITHMS

It is becoming almost impossible to design and test large systems that depend upon complex software algorithms without the aid of simulation. Using VisiSoft, the algorithms written in VSE can be placed directly in GSS simulations. Companies that use VisiSoft have the significant advantage of using GSS as a design tool for new algorithms and also a test tool for existing algorithms. Creating test cases that repeat complex conditions is very difficult without the aid of simulation. So when problems are encountered in the field, simulation can be used to determine the root causes, redesign fixes, and run further tests without creating live tests, generally a dramatic way to cut scarce personnel time and costs.



Chapter 16. Very Large Scale Systems

This chapter describes cases that occur with large scale systems and complex software. In engineering, one learns to investigate the *limiting cases*, producing limiting factors that help one to determine the best technological approach to solving problems in design architecture and language. In this chapter, we investigate extremely large software systems that stress many aspects of the software development and product upgrade process.

SINGLE PROCESSOR SYSTEMS

In this section, we are looking at large scale software systems designed to run on a single processor. Multi-Processor and Parallel processor systems are covered in following sections.

Motivation

The new paradigm described in this book was driven by the need to develop large scale simulations of communication and control systems, simulations that would have to be run very fast - on parallel processors under a single operating system. As indicated in Chapter 5, this led to the “separation principle,” [55]. This approach allows one to track software module independence and automatically allocate processors to processes at run-time on a large parallel processor.

As we have shown in prior chapters, the separation principle also provides the basis for engineering drawings of software, with a one-to-one mapping from the drawings to the code, a true form of *software architecture*. Prior to *VisiSoft*, software architecture did not exist, an observation that should now be apparent to the reader. The analogy between current programming approaches, and architects in other fields trying to produce designs without drawings, should also be apparent.

Current Pertinent Comparisons

Operating systems have always been difficult pieces of software to build, going back to the days of OS-360 and the collapse of the MULTICS project. Time response requirements on speed, multiple tasking coupled with handling large numbers of events in real time, and managing large numbers of distributed databases is difficult enough. But sheer size - Microsoft claims that Windows is up to 50 million lines of code - leads one to question the lack of an architecture. Having 4000 programmers writing code presents even more questions.

It is pertinent to make a comparison of two companies in the Seattle, WA area. Anyone who has been inside a hanger at Boeing where large aircraft are assembled has to be amazed at the size of the hangers and the complexity of the assembly and testing process. But one had to be equally amazed at the size of the drawings wrapped around the walls of these hangers - multiple stories high - with rolling catwalks to review them. Most everything is on computer terminals now. But without these CAD systems and drawings, one could not begin to understand how it all fits together. Now imagine taking the drawings away and having everything described in a language - as is done at Microsoft.

This is not an absurd analogy. The Windows operating system happens to offer an extreme case. But there are many systems with a few million lines of code. Having used *VisiSoft* for years, one cannot envision controlling the architecture of a system with just 10,000 lines of code without a good architecture and corresponding drawings.

After experiencing the development of large scale systems (more than a million lines of code) using *VisiSoft*, one learns that simply invoking an architecture by an experienced architect can cut the number of lines of code by whole numbers. The combination of engineering drawings, high level languages, and large data structures and rule structures - that are controlled hierarchically and easy to follow - can increase productivity by an order of magnitude when upgrading and enhancing a large software product.

Managing Libraries Instead of Managing Code

Above 1,000,000 lines of code, management of dynamic distributed databases, fielding and scheduling real time events, and complex 2D and 3D graphical user interfaces translate into managing large numbers of libraries. An example is managing dynamic lists. If programmers are building different linked lists for different applications, they are likely being mismanaged. In simulations and real time systems that move huge amounts of data, linked lists appear in many higher level modules. In some cases, the same programmer may decide to tailor more than one of these for different functions, usually because they are “pressed for time” to get out the release. They don’t have time to build one that can be used three or four times.

Now multiply this by just 20 programmers and one may have on the order of 50 linked lists, each being debugged and tested separately. It is likely that this number could be reduced to three or four library modules that are bug free as well as extremely fast. This requires a library management facility to ensure that everyone knows what is available, and that rebuilding one of these is unacceptable practice.

When building a large scale system, library development and management is critical. When building the next system, it is even more critical, since the next system is likely to be larger, but can be put together and tested faster given a huge set of reusable libraries.

Having engineering drawings of the software enforces the practice of using libraries. This is because an architect can specify libraries to be used by the coders, and can then check the module drawings to ensure no one is rebuilding an existing module - one that is just a tiny bit different than one on the shelf. Most often, the tiny difference can be incorporated into an existing library without changing the way it works for the current users. Then it is still the same library with a slightly new feature. If it must be slightly different from the existing module from a user standpoint, the person responsible for the existing library module is the best person to copy it, rename it, modify it, and make it available as a new module. It will likely be built and totally debugged in less than 10% of the time.

Libraries contain reusable modules that can be shared by huge numbers of programmers. Documenting library functions and how they are used is part of the library management function. The amount of time taken to organize, document, and distribute libraries is paid back in whole number multipliers, from the first time they are used.

PARALLEL PROCESSING

Hardware designers have succeeded in producing parallel and distributed processor computers with theoretical speeds well into the teraflop range. However, the practical use of these machines on all but some very special problems is extremely limited. The inability to use this power is due to great difficulties encountered when trying to translate real world problems into software that makes effective use of highly parallel machines. This has been described by numerous authors over many years, see for example [70], [8], and [67].

COMMERCIAL MARKET REQUIREMENTS

In the commercial marketplace, speed benefits gained using a parallel computer must sufficiently outweigh the cost to develop and support the software. If not, then real commercialization, based upon solid economics, will not occur. These economic goals will be achieved if the following requirements can be met:

1. Subject area experts who understand the problems to be solved must be able to describe them easily and directly to computers without concern for parallelism, or even prior knowledge of computer programming.
2. The software must be generated automatically to take full effective advantage of the inherent parallelism of the problem on a Massively Parallel Processor (MPP).

These two requirements are tightly interrelated. The subject area expert should not care whether the problem is being solved on a single processor machine, or one with hundreds of processors. The run-time software must be generated to make effective use of the available parallelism of the host machine, adapting to changes in the environment, a very tedious but mechanical process.

REQUIREMENT FOR SPECIAL PROGRAMMING SKILLS

Current approaches to solving problems on parallel processor machines have not, in general, overcome these two barriers. Problem description for parallel - as opposed to single - processing generally incurs a huge cost increase for all but a few special cases. This is compounded by the fact that the problems requiring large processor power are themselves complex, and best understood by subject area experts.

For example, a communications engineer trying to design a specific set of algorithms, to implement a very complex set of protocol standards, has difficulty just describing his problem using graphic diagrams with plain English text. To constrain him to describe his problem in an esoteric programming language is difficult. To force him to learn the language of a system programmer, i.e., the operating system, is unlikely. To further burden him to describe his problem so that it runs efficiently on a parallel computer makes the approach intractable.

One is then led to an approach that augments the engineering staff with parallel processor programmers who perform problem translation for the computer. However, it is well accepted in most engineering departments that, when programmers are used to translate an engineer's problem to a computer, problem solution becomes a process whose length increases exponentially with problem complexity. Finally, translation onto a parallel processing machine currently requires very special programming skills that are commensurably scarce and expensive.

This is why engineering departments invest heavily in Computer-Aided Design (CAD) tools that they interface with directly - on their own terms. These CAD tools provide interfaces that are tailored to their problem and automatically generate highly efficient computer code. We believe that this is the solution approach to be taken toward commercialization of parallel computing.

HARDWARE FIRST

Solutions to the parallel processing problem tend to skip over the software piece of the problem, going from application requirements to hardware architecture. (The word *architecture* implies hardware in the parallel processing literature. The words “software architecture” do not appear.) Software is not much more than an afterthought relative to the size of the hardware design effort. This approach, illustrated in Figure 16-1, is termed *software bypass*.

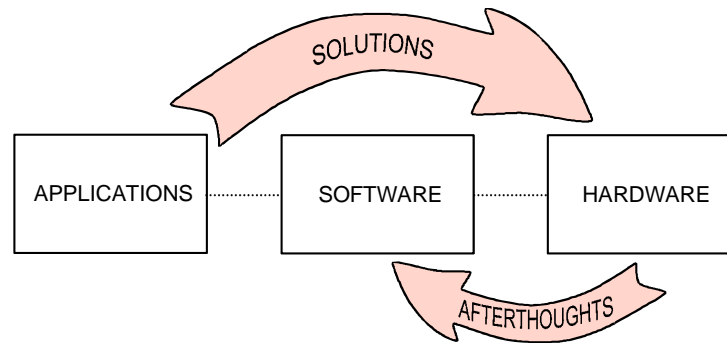


Figure 16-1. Software bypass - designing the hardware first.

Subject area experts who want to use parallel computers cannot simply enter their problem specifications into a piece of hardware. They must first write the very complex software required to control parallel processor hardware. Without knowledge of the special operating systems and languages for parallel computers, these experts typically turn to programmers to do the job. Programmers see the chance to increase their value by learning how to be parallel programmers. Their interest is in learning deeper specializations to broaden their higher-paying job opportunities. This cycle of thinking is at odds with commercial market requirements.

USE OF ABSTRACT REPRESENTATIONS

Certainly there are many uses of abstractions when building models of highly complex systems and their environments. One could not perform simulation without abstraction of reality into models that run on a computer. The General Simulation System (GSS), [45], provides for ease of abstraction where complex processes that may be spread across all of the entities in a system are represented in a single list. GSS contains a library of high speed list management facilities that eliminate the need for the modeler to develop linked list software, a basic abstraction in modeling. However, one must consider the trade offs between time and cost of development as well as speed and memory utilization at run-time.

With today's parallel processors, memory utilization is not an issue. It is difficult to conceive of a problem where the amount of memory on a large parallel processor computer presents a limitation. Using conventional techniques for parallel processing, the trade is usually between development time and running time, given resource constraints in dollars. This leads to decisions on how models are represented.

The choice is usually between the way one deals with abstractions, and typically ends up with substantial hand tailoring of code to the parallel processing environment. This implies a huge effort in development, resulting in significant time and cost, to use parallel processors. More importantly, the abstractions required for parallel processing make it difficult for a modeler with subject area expertise to understand the code.

THE INHERENT NATURE OF SYSTEM DECOMPOSITION

As systems are designed to be more user-friendly and adapt to their environment with greater effectiveness, they become more complex. To deal with a high level of complexity, designers must partition systems into modules that operate independently, minimizing the shared interfaces. If module interfaces are designed for maximum isolation, they incur a minimum transfer of information. This maximizes the ratio of internal processing to interface processing, which in turn maximizes their measure of independence. This is the type of software architecture required for effective use of parallel processing. Given a high degree of module independence and inherent parallelism, many applications have still failed to achieve a high degree of efficiency in parallel processor utilization. This is because current software approaches cloud this level of architecture.

The two most prominent parallel processing companies in the early 1990s, Kendall Square Research (KSR) and Thinking Machines Corp. (TMC), failed due to lack of good software environments for both developing and running applications. There are a number of reasons that no software environment has yet to crack the problem. We believe that the two most important reasons are:

- (1) Decomposition of a large software system is an architectural problem, and the architecture of a system of independent modules is best described graphically (like hardware) - not using a language;
- (2) Software architectural design methodology and supporting technology have not been tied to the requirements of efficient scheduling and assignment of processors to processes during run time.

After one gains a good understanding of the software side of the parallel processing problem, it becomes clear that the *language environment* must be designed to support the *architecture environment* as well as the requirements for understandability and independence of the detailed implementation. This has major implications on *scoping the size* and *controlling the hierarchies* of independent modules. At least as important, the architecture environment must serve to optimize the scheduling and assignment of processors to processes in the *run-time environment*. The VisiSoft solution solves both problems.

PERTINENT CONSIDERATIONS

Future survival depends upon the speed with which one can deal with increasing complexity.

THE IMPACT OF SPEED AND COMPLEXITY ON SURVIVAL

The things we take for granted today would have boggled the minds of people just 100 years ago. Looking back 1000 or 10,000 years is awesome. Which way would any of us prefer to live? Who is better prepared to survive? The answer to the first question is generally obvious. The answer to the second requires more consideration.

The U.S. is learning that there are many faces of survival. The days of firearm versus bow and arrow are long past. Yet a high speed aircraft with smart missiles may not help preserve our own infrastructure when attacked by terrorists. The approach to survival is taking on a different meaning than historic war. The enemy situation is becoming much more complex. Accurately predicting what an adversary may do depends upon how much time he has to think, communicate, and take action. The problem of defending the U.S. is being redefined in light of the increasing need to deal with speed and complexity as we endeavor to survive.

Dealing With Increasing Complexity

Anyone familiar with the history of mathematics knows the motivations leading to the progression of numbers. It started with “whole numbers” or *integers*, and progressed to *signed integers*, then to *fractions* and *rational numbers*. It continued to *real numbers*, *imaginary* and *complex numbers*. Each step covered a more complex realm - not by imagination, but by necessity.

There is more to this progression than just the increase in complexity. Each of these extensions is still referred to as a number. And each encompasses the prior. Real numbers are a subset of complex numbers. More importantly, many of the laws and transformations still apply as we move up the scale of complexity. Their interpretations are simply extended to be more general. This allows us to deal with jumps in complexity.

Selecting The Most Convenient Coordinate System

As we continue to move up the food chain of numbers and mathematics, we can group numbers into *vectors*. The position of a body in space can be described by three numbers depending upon the coordinate system we choose. And we learn in higher levels of mathematics and physics, particularly in electro-magnetic theory and partial differential equations, that problems can be solved more easily if we select the right coordinate system. For example, when a particle moves in a spherical orbit, it is much easier to describe its motion in spherical coordinates. Cartesian coordinates will work, but it takes longer to solve the problem.

Selection of the most convenient coordinate system is typically taught under the topic of *separation of variables*. One learns that the separation principle can be used if the variables form an *independent set*. The property of independence can be verified using specified tests. The concept of choosing the best coordinate system and the property of independence are the important principles one can apply when dealing with complexity in a constrained time environment. We will make use of these concepts.

Einstein introduced the use of *tensors* to deal with the increasing dimensions of time, velocity, and acceleration. Control system engineers developed the *state vector* to account for the many degrees of freedom required to characterize complex dynamic systems. The *state space* framework has been shown to be the most general representation of a dynamic system, see [4], and [39]. Providing a framework for problem description was not the only benefit of the state space approach. It also afforded a framework for developing faster solutions to problems that could run for days on the computers of the time.

FRAMEWORKS FOR REPRESENTING COMPLEX DYNAMIC SYSTEMS

In a competitive time-constrained environment, time (speed) is the most important factor. If two sides develop the same capability, the one that gets there first is likely to be the one that wins. When building tools to help people solve design problems or make complex planning decisions, time enters into the picture in at least two major ways.

- Development Time - the time it takes to develop the tool
- Solution Time - the time it takes to get a useful solution from the tool

One can imagine a great tool for solving a problem. But one must answer the question - can we get it built in time to accomplish our goal? Or, more importantly, will it produce valid answers *fast enough* if we get it built? Of course cost and risk are also major factors. However, time is usually of the essence.

Automating The Representation Process

As we have indicated in the early chapters, electronic circuit designers developed automated tools for solving complex systems of nonlinear differential equations required to represent digital waveforms in the time domain. These Computer-Aided Design (CAD) tools allowed engineers to describe large networks topologically and write FORTRAN-like equations describing nonlinear functions. Programming skills became unnecessary. The code needed to generate and run simulations of very large networks was generated automatically. This afforded a huge leap in design productivity. It enabled the design of huge complex networks leading to integrated circuit design.

CAD system development became a business for many, including the principals of VSI. Two systems were developed, one for continuous system modeling (e.g., for digital circuit design), and one using a discrete-time framework (for the design of signal processing systems). The second used sampled data principles to reduce computation time. An underlying *state space* framework supported both products.

For large networks, the number of state variables runs to thousands. Solving worst case design problems involves multiple optimization runs of thousands of simulations. Each simulation has to solve the optimal control problem, involving thousands of nonlinear differential equations. Speed and accuracy are the driving forces in designing these systems. If it takes a computer days to get a design, only one or two test points are produced in a week - not very attractive.

Capitalizing Upon General Principals

State space is used because it provides the most convenient framework for solving any type of dynamic problem. The general form of the solution holds for any set of independent state variables. This allows for the development of generalized methods, e.g., optimal sparse matrix inversion and describing functions, to solve nonlinear problems fast while ensuring algorithm convergence. The end result is to solve huge problems in minutes. However, this approach requires formulating problems in a mathematical framework.

Facing Totally New Problems

Models built using VSI products prior to 1982 were formulated mathematically, i.e., using vectors, matrices, and systems of equations. This approach allowed the solution to be derived automatically and solved very fast. By 1982, this approach was recognized to have severe limitations when modeling communications or control systems involving algorithmic decision processes. Clients wanted to describe their problem using more general state concepts, and be able to write conditional statements within the system of equations. It was determined that these types of decision processes could be handled using the discrete event approach originally developed by Gordon in 1961, see [41] and [42].

A MORE GENERALIZED PROBLEM FORMULATION

In 1982, discrete event simulation was analyzed. The motivation was high because of the requirement for writing decision algorithms into the models. Users wanted to break up systems of equations and embed English-like conditions and rules, e.g.,

```
IF THE MESSAGE_TYPE IS CONTROL, THEN ... ,  
ELSE IF MESSAGE_TYPE IS DATA, THEN ... .
```

Additionally, there were complaints about the inability of existing discrete event simulation products, e.g., GPSS, SIMSCRIPT, and SLAM, to solve our client's problems. The major complaints were lack of scalability (inability to deal with increasing complexity) and excessive simulation run-times. This led to an investigation of the deficiencies of the other products in the market, as well as an analysis of how to formulate the basis for general solution.

At first it appeared difficult to derive a mathematical framework to support this new requirement. This caused concern about the ability to justify design decisions without a formal framework. We appeared to be leaving the world of mathematics. Time steps were determined by the modeler in terms of scheduled events. This led to the development of a state space definition of discrete event systems. The concept of a generalized state vector and state space definition of a GSS model was described in Chapter 5. The differences and likenesses of mathematical and rule oriented formulations are compared in *Simulation Of Complex Systems*, [27].

Facing The Speed Issue

Because of the excessive running times of competing products (some critical simulations were taking 5 to 7 days to run a 2 hour scenario), it was determined that if a new product was developed, it must be able to run on a parallel machine. The experience of the VSI principals in computer design, parallel processing, and the knowledge of how chips were evolving to support fast computing methods led to an approach that would take advantage of future hardware technology.

As indicated in earlier chapters, parallel processing imposes the requirement that two or more processes must run concurrently on separate processors. This implies that concurrent processes must be independent. The property of independence implies that the processes share no data. This led to the decision to separate data from instructions so the independence property could be tracked. As previously described, the design of GSS was launched in 1982. It called for a connectivity matrix to determine what processes shared what data. Then when allocating processes to processors, the connectivity matrix could be used to determine if a process can run concurrently with those already running.

Independent Instanced Models - Modeling Made Easy

The separation of data from instructions led to the ability to produce engineering drawings of models, where the lines connecting models determine the independence or lack of it. This allows an architect to visually inspect the drawings and determine the independence of a model relative to other models. These concepts have led to the independent instanced model as part of the GSS environment. This allows a modeler to build a single model along physical lines, just like building a single piece of equipment. This model can then be instanced many times, automatically, in a simulation. This paradigm makes it easier to develop models on a large parallel processor than by using current methods on a single processor. This capability has been implemented as part of prior efforts.

OVERVIEW OF THE USER INTERFACE TECHNOLOGY

When the GSS environment for discrete event simulation was designed, the two major issues addressed were: (1) the difficulty of building valid models; and (2) the time to run a realistic scenario. The difficulty in building valid models was due to the complexity of the software. Run time may have been reduced by parallel processing, but the investment was huge and risky. To address these issues, the CAD approach was developed that led directly to the effective use of highly parallel processors. We note that software applications are considered easier to implement on a parallel computer than discrete event simulations because in the simulation environment, one must: (1) synchronize each process with the main simulation clock; and (2) ensure synchronized data coherency to meet validity requirements. From this standpoint, *the software problem is a subset of the simulation problem.*

Separation Of Data From Instructions For Efficient Processor Allocation

In software, separating data from instructions violates the OOP rules. In hardware design, this paradigm is the norm as described in Chapter 5. Data and instructions are separately stored and managed on today's chips. This is an essential software paradigm for effective use of parallel computers, where one has to allocate processes to processors efficiently. This implies knowing which processes can run concurrently, which implies that they must be independent. Independence is effectively determined by whether or not they share data. If allocation is to be done automatically, the allocation manager must have the information on who shares what data. The technology described here is built upon this concept. The most significant paradigm shift in this development environment is the separation of data from instructions.

The resulting properties of the technology described here provide enormous benefits for parallel processing software design. First is the ability to represent software graphically, with a one-to-one mapping from the drawing to the code. Second is that software architectures can be designed and reviewed from an engineering standpoint to determine module independence. Third is the resulting connectivity map of what processes share what data. Fourth is what processes reside inside what modules. If modules are independent, then processes within those modules are best migrated to the same processor. This information is stored in our run-time as well as development databases. It is this information that provides our ability to optimize the allocation of processes to processors to maximize run-time efficiency. These benefits are best described by an example.

A Large Simulation Example

We will use the Multi-Switch Simulation (MSS), a large scale communications simulation to describe our CAD facilities. MSS contains nine modules, including circuit, packet, and ATM switch modules. The ATM switch module, shown in Figure 16-2 along with the ATM_LINK module, is a *hierarchical module* containing seven submodules.

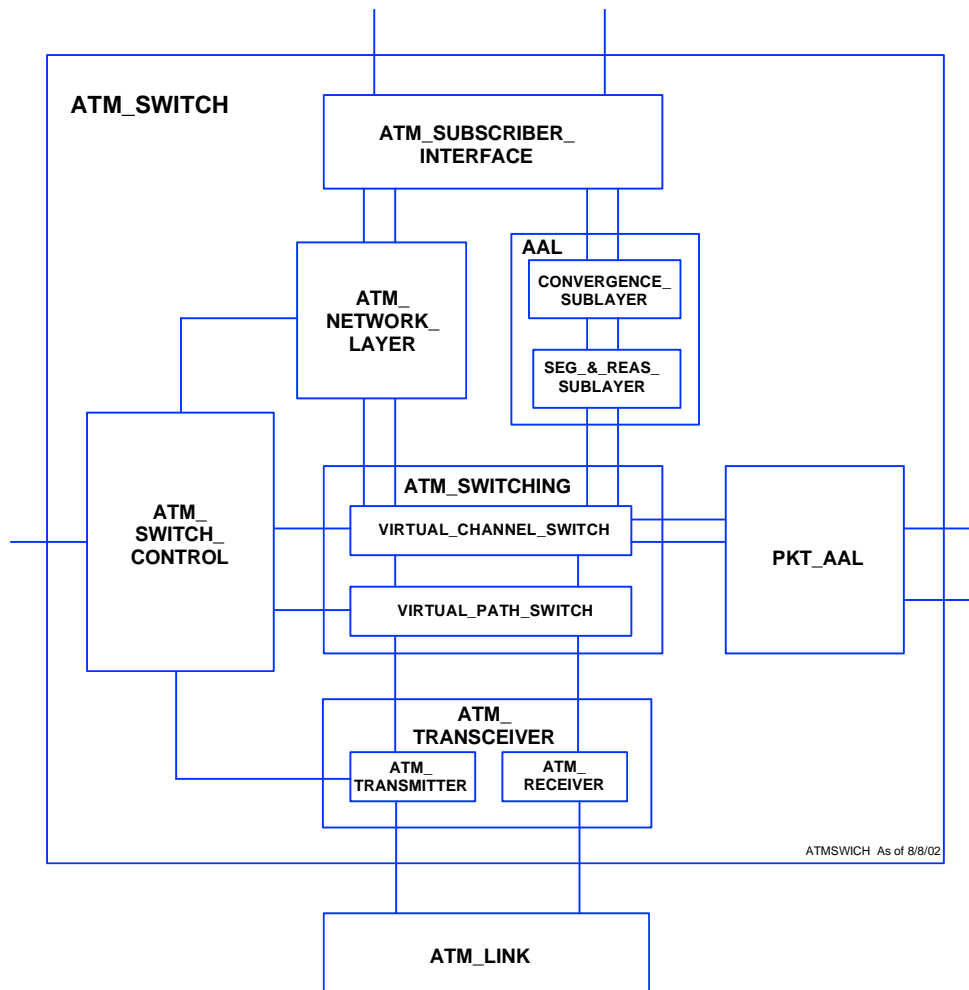


Figure 16-2. ATM_SWITCH and ATM_LINK modules.

The AAL submodule is also hierarchical, containing the CONVERGENCE_SUBLAYER and SEG_&_REAS_SUBLAYER modules. The SEG_&_REAS_SUBLAYER module, shown in Figure 16-3, is also hierarchical, containing the SR_PROCESSOR and SR_QUEUE modules. These are each *elementary modules* because they contain *primitive elements*, namely *resources* (ovals) and *processes* (rectangles).

Resources are composed of hierarchical data structures. An example is shown in Figure 16-4. Resources are used to describe the state of a module at any instant of time. These are shared by the processes that have connect lines drawn to them. Processes are composed of hierarchical rule structures, e.g., SR_PROCESSOR shown in Figure 16-5 (it shows 7 rules).

Processes are used to transform modules from state to state. These processes are not *tasks* as in a “multi-tasking” operating system (and therefore not UNIX “processes.”) In GSS, *processes that are scheduled are parallel threads*. A simulation can run as a task (a UNIX process). VSE and GSS also provide for intertask control and communications at the task level so multiple simulations can run and interact as separate tasks.

We have taken significant departures from existing software concepts to automatically generate code to use parallel processor resources effectively, without concern by the user. The first departure is separation of architecture from language. Design of module architectures is done in the *architecture environment* of VSE and GSS, not the *language environment*. In the architecture environment, the user determines graphically what resources have access to what processes as shown in the drawing in Figure 16-3. To do this requires the second departure, namely the separation of data (resources) from instructions (processes). Process independence can then be determined simply by looking at the lines interconnecting processes and resources across modules in the architecture, i.e., *module independence is determined by the architectural drawing*.

Discrete event simulation has provided us with a view of software design and parallel computing that is not afforded in other technologies. First, GSS processes are *scheduled* - refer to Figure 16-5 where SR_PROCESSOR is scheduling itself in the first rule. When the MSS simulation runs, many thousands of processes are in the schedule at any instant of time. Of these, more than half can be scheduled to run at the same time, e.g., the current time. These are candidates for running concurrently. Second, we distinguish between software abstractions (that prevent concurrent processing) and direct representations of the real-world instructions that can run concurrently.

For example, the SR_PROCESSOR module makes use of a number of utilities (connections between a process and a called utility are indicated by circled letters and numbers). These utilities (green borders), e.g., SR_QUEUE, help to save memory by having a common set of instructions serve each instance of the switch. Some of the data are reused also, but this is generally small compared to the data stored by instance - data that can reside separately with each instance. When memory was expensive, this small memory savings was justified. When running on a parallel processor, with a large memory model, this major bottleneck is unjustified and all the circled connections can be removed. This renders the module highly independent and reduces complexity at the same time!

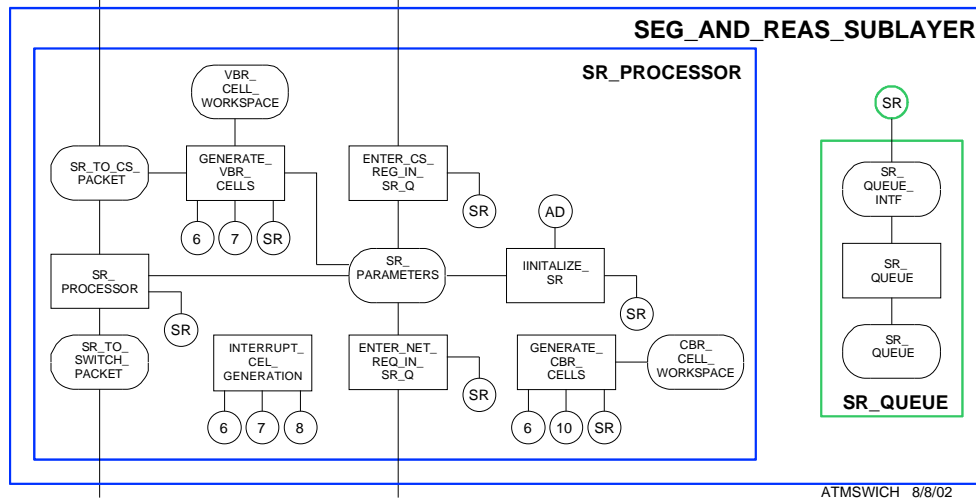


Figure 16-3. SEG_&_REAS_SUBLAYER module.

RESOURCE NAME :		MESSAGE_FORMATS	
STANDARD_MESSAGE			
1	SYNC_CODE		CHARACTER 5
	ALIAS	VALID_CODE	VALUE '10101', '01010'
1	TYPE		STATUS FORMAT_A FORMAT_B
1	CONTENT		CHARACTER 46
FORMAT_A REDEFINES STANDARD_MESSAGE			
1	PAD		CHARACTER 13
1	HEADER_A		
	2	MESSAGE_PRIORITY	STATUS FLASH PRIORITY ROUTINE
	2	ORIGIN	INDEX
	2	DESTINATION	INDEX
		ALIAS BROADCAST	VALUE 0
1	BODY_A		
	2	BODY_LENGTH	INTEGER
1	TRAILER_A		
	2	MESSAGE_NUMBER	INTEGER
	2	TIME_SENT	REAL
	2	TIME_RECEIVED	REAL
	2	ACKNOWLEDGMENT	STATUS RECEIVED NOT_RECEIVED
	2	LAST_SYMBOL	CHARACTER 2
		ALIAS TERMINATOR	VALUE '\\', '//', '<<', '>>'
FORMAT_B REDEFINES STANDARD_MESSAGE			
1	PAD		CHARACTER 13
1	HEADER_B		
	2	SOURCE	INDEX
	2	SINK	INDEX
1	BODY_B		
	2	CONTENTS	CHARACTER 42

Figure 16-4. A resource - a hierarchical data structure.

```

PROCESS:          SR_PROCESSOR

RESOURCES:   SR_TO_CS_PACKET          INSTANCES:   NODE
              SR_PARAMETERS
              SR_QUEUE_INTF
              SR_TO_SWITCH_PACKET

PKT_SR_PROCESSOR
EXECUTE GET_SR_MESSAGE
EXECUTE PROCESS_SR_MESSAGE
IF QUEUE_STATE IS NOT EMPTY
    SCHEDULE SR_PROCESSOR
        IN PROCESSING_TIME MICROSECONDS USING NODE
ELSE SET PROCESSOR_STATUS(NODE) TO IDLE.

GET_SR_MESSAGE
SET SR_QUEUE_INTF REQUEST TO DEPART
CALL SR_QUEUE USING NODE

PROCESS_SR_MESSAGE
IF PACKET_TYPE IS A CELL
    EXECUTE PROCESS_CELL
ELSE IF PACKET_TYPE IS A REQUEST
    EXECUTE PROCESS_REQUEST
ELSE EXECUTE INVALID_PACKET_TYPE.

PROCESS_CELL
MOVE SR_QUEUE_INTF MESSAGE TO SR_TO_SWITCH_PACKET
IF SR_TO_SWITCH_PACKET DESTINATION IS EQUAL TO NODE
    EXECUTE CHECK_PAYLOAD_DEST
ELSE IF SR_TO_SWITCH_PACKET SOURCE IS EQUAL TO NODE
    EXECUTE CHECK_PAYLOAD_SOURCE
ELSE EXECUTE INCORRECT_NODE.

CHECK_PAYLOAD_SOURCE
IF PAYLOAD_TYPE IS USER_VOICE
    EXECUTE PROCESS_CELL_VOICE_SRC
ELSE IF PAYLOAD_TYPE IS USER_DATA
    EXECUTE PROCESS_CELL_DATA_SRC.

CHECK_PAYLOAD_DEST
IF PAYLOAD_TYPE IS USR_VOICE
    EXECUTE PROCESS_CELL_VOICE_DEST
ELSE IF PAYLOAD_TYPE IS USER_DATA
    EXECUTE PROCESS_CELL_DATA_DEST.

PROCESS_CELL_DATA_SRC
EXECUTE GET_MESSAGE_INFO_CELL
INCREMENT MESSAGE CELLS_TRANSMITTED
EXECUTE UPDATE_MESSAGE_INFO
IF MESSAGE CELLS_TRANSMITTED IS EQUAL TO MESSAGE CELLS_TO_TRANSMIT
    EXECUTE GET_NEXT_MESSAGE.
CALL ENTER_USER_REQ_IN_VC_Q USING NODE

... (This process is incomplete - 2 additional pages are not shown!)

```

Figure 16-5. A process - a hierarchical set of rules.

To insure independence of modules, a set of architectural design rules has been developed that can be enforced automatically as the designer builds modules graphically. This involves viewing a module as an *N-port module* as used in electronic hardware design. By limiting the number of lines (wires) at a port to two, the independence of modules is ensured. Note that we have not considered any aspects of coding, which in VSE or GSS is confined to the language environment. We have only analyzed the module architecture - graphically! These design rules assure ease of module understandability and independence, and therefore real reuse. They are the major reasons we have been able to build and validate the world's largest simulations at very low cost. This same technology is ideally suited to make effective use of highly scalable parallel processor computers.

Another departure from typical software is the integrated management environment of VSE and GSS that completely tracks the architecture behind the scenes and contains the databases to determine both spatial and temporal independence at run-time. Modules are tracked through all of the hierarchical levels needed by the designer to control design complexity. Every resource and process is tracked relative to what processes have access to what resources within multiple module instances. This database can be used to adaptively manage the allocation of parallel processor resources during run-time based upon knowledge of module instance independence at any level in the hierarchy. Load balancing can be achieved concurrently through selected instance migration. This critical information is not available anywhere else!

We will now relate the number of module instances to opportunities for parallelism. As the top level modules, e.g., a switch, take on higher degrees of complexity, they become significant opportunities for highly efficient parallel processing. If the switch is modeled along physical lines, its physical counterpart operates concurrently with its neighbors. Therefore, independent module instances in a simulation can also run concurrently in a parallel processing environment. Such instances are not limited to simulation, but exist frequently in real-time control and communication systems.

Based upon this concept, our hypothesis is as follows: As the number of instances of a complex independent module increases, the number of parallel processors that can be used effectively increases proportionately, just due to the independent module instances. Similar opportunities for effective use of processors can also be obtained within a top level module instance, down to the process level. This is because of the hierarchical design and resulting scope of a VSE or GSS process.

For example, the ATM_TRANSCEIVER within the ATM_SWITCH in Figure 16-2 can have 20 instances (one for each port), all tied to the same instance of a switch. A scenario of 100 switch instances can invoke a total of 2000 ATM_TRANSCEIVERS. We can envision many instances of subscribers as well as other packet and circuit switches running concurrently, interfacing with each other through links or gateways. Each of these instances can run concurrently since almost all of the processes and resources are *interior* to the instance and therefore *independent* of the other instances.

Quantifying The Importance Of Software Architecture.

To better understand this typical architectural phenomenon, consider the modules in Figure 16-6.

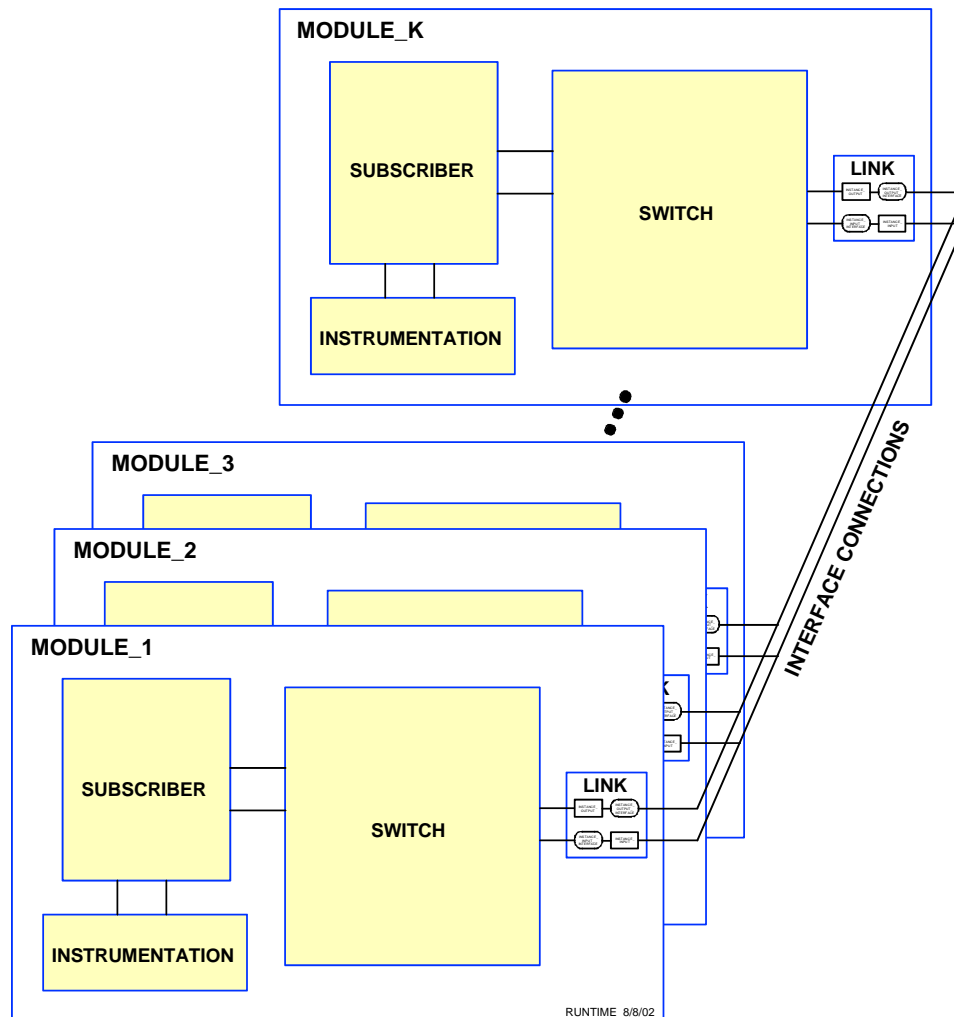


Figure 16-6. Independent *instanced* modules connected by an interface.

The top level modules in Figure 16-6 are drawn alike for simplicity, but in fact may be different types or instances of the same type. As an example, we will consider an MSS simulation with 100 circuit switches, 50 packet switches, and 50 ATM switches. Consider that each instance of each switch is part of a single module along with its corresponding subscriber submodule instance that generates and receives voice calls and data messages and files, and its instrumentation submodule that takes measures of traffic. These large submodules are the largest part of each module. A link interface submodule also exists connecting each top level module. Except for the two processes connected from each module to the interface, all other processes in each module are independent of those in any other module, i.e., they share no other resources between modules. This is done by design - *of the software architecture*.

OVERVIEW OF THE RUN-TIME TECHNOLOGY

Significant work has been done by VSI on prior projects toward development of the required run-time technology. This work covers the use of the module architecture knowledge described above as well as knowledge of the independence of individual processes at the module boundaries to determine what processes can run concurrently. This work includes development of the protocols required to ensure data coherency of resources shared across module boundaries and used by processes in different processors. It includes the synchronization of scheduled processes running on separate processors in a simulation. It provides for controlled variations in synchronization that ensure validity of results of a simulation - something not provided by other approaches, e.g., the Time-Warped Operating System, and its derivatives (e.g., SPEEDES). It provides for optimal ordering and scheduling of p-threads.

Figure 16-7 below provides a top level view of the design for the VSE/GSS run-time environment for an MPP system. In addition to the Process Scheduler, there is a Processor Allocator to allocate processes scheduled at the current time (or within a pre-defined ΔT_{max} in a simulation) to the available processors. This design uses standard OS level calls to assign parallel threads (p-threads) to processors. This provides the ability to allocate specific processes to specific processors, including the ability to reallocate processes to processors for dynamic load balancing if necessary.

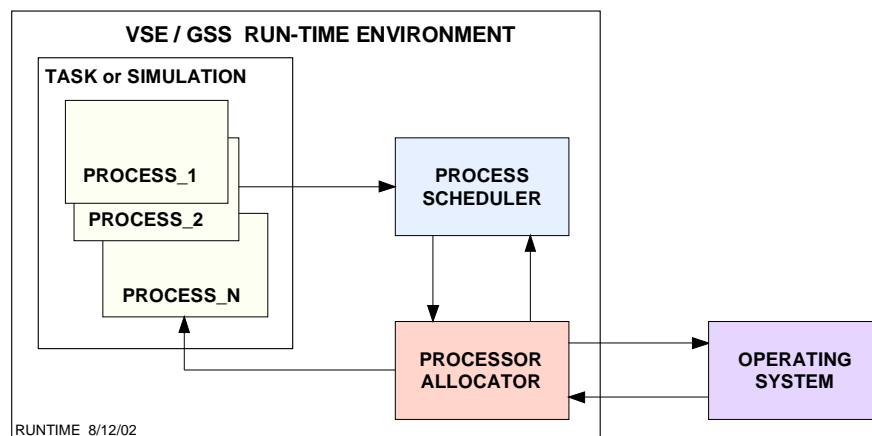


Figure 16-7. Connection between the VSE process scheduler and the processor allocator.

There are additional mechanics of this environment to be characterized, e.g., the nature of the dynamic changes to the schedule versus the state at time T. This affects the algorithm design for optimal ordering in minimal time. Instanced modules create special submatrices of the connectivity matrix that are independent. These become candidates for quasi-independent queue management, potentially in separate processors. VSI's experience in discrete event simulation for the past 20 years provides significant knowledge of solutions for these types of problems. In addition, processor load balancing must be considered in more detail, but this has been the subject of much prior research, both at VSI and elsewhere. Finally, marrying this new technology to hardware is an architectural design issue. VSI has worked with many hardware vendors in the past, and is prepared to work with more in the future.

Summarizing The Importance Of The Software Environment

Given applications with a high degree of inherent parallelism and very efficient parallel computers, their effective use comes down to three major factors. First is ensuring that full advantage can be taken of the inherent application parallelism - a software design problem. Second is balancing the load - a run-time software problem. By separating data from instructions and using the visual development environment that PSI has already developed, the software architectural knowledge exists to do both well. The third and most important factor is making it easy for the subject area experts to describe their problem, without having to twist it into a special computer language. VSI's success in CAD tools for building very complex discrete event simulations and software tools has already demonstrated the ease with which this is done. This has built confidence in the ability to bring large scale parallel processing power into the mainstream of computing via *ease of use* - the winning "WinTel" approach.



Chapter 17. MANAGING SOFTWARE LIFE CYCLES

This chapter addresses the software life cycle management problem from a perspective of the future software industry, and how competing companies must manage in providing both tailored systems and software packages to their customers. We consider both cases as producing products that require support for new releases.

There are many texts and papers on the subject of software management. They generally describe the life cycle in the modern context, as we have in Chapter 5, implying a continuing cycle or incremental approach (an early version was prescribed in 1984, [24]). They also provide ample guidance as well as policies, procedures and standards. We will cover neither step-by-step management methods nor equivalent details here. Instead, we indicate the types of approaches one must consider when managing a software product in a rapidly changing competitive environment, and how one can adapt management approaches to the new technology paradigm presented here.

BOUNDING THE ISSUES

Depending upon where one sits in an organization, there will be a multiplicity of views of the problem of managing software life cycles. One may be involved in an in-house project for a small system that will live a relatively short life and require just a few programmers. Alternatively, one may be a senior executive in a large software company considering an investment involving hundreds of people to develop and support a new product to be sold internationally. Clearly, there are a large number of project sizes in between these two extremes. This chapter is aimed at the mid to upper part of this scale and beyond.

Quality Versus Productivity

The Capability Maturity Model (CMM) of Carnegie Mellon's Software Engineering Institute (SEI) is aimed at improving software quality, particularly in the eyes of U.S. Government buyers, see [85]. It is known that this approach may cause productivity to suffer as one works to increase quality as measured by the CMM. This is in contrast with the highly acclaimed and proven quality control approaches as described by well known experts, e.g., W. Edwards Deming, see [34], and Joseph M. Juran, see [54].

Although the SEI-CMM approach claims to be compatible with the Deming/Juran approaches, it is basically different. If one follows the Deming/Juran approach, productivity and quality should rise together. Also, in the Deming/Juran view, quality does not emanate from management direction or inspection; it comes from design. To do a proper design requires many of the same functions, but they are driven from a different direction, one that fosters productivity improvement. For another view on the advantages and shortfalls of CMM, the reader is referred to Bach, [6].

Our objective is to improve productivity through technology innovation while maintaining - if not improving - quality. Our definition of quality is represented in [23] and [24]. A compatible definition of productivity is provided in [1]. More specifically, by using the new technology described in prior chapters, we have witnessed great reductions in the required intensity of management oversight as well as in the density of programmer activity to achieve similar if not better quality outcomes. This has led to much higher measures of relative productivity.

Management Versus Technology

In a rapidly changing competitive environment, management's most important task is to instill the drive for innovation, to improve the quality-productivity combination. When dealing with innovation to garner real improvements, managers must make cautious use of conventional wisdom. As stated by Christensen, [29], this may imply *not* listening to one's big customers. This is because innovation changes the framework for measuring what's *best*.

A good example of this phenomenon relates back to software companies that specialized in accounting packages in the early 1980s. These companies typically worked with their customers in an attempt to come up with generalized *packaged systems* for accountants to automate their bookkeeping. Then a little software company with no such clients - and hardly any management - came up with a "spreadsheet" (VisiCalc). It wasn't long before accountants started building their own spreadsheets, and most of the software companies building accounting packages went out of that business.

Good management is hard requirement. But a great new technology can make an order of magnitude of difference - independent of management. One could argue that really good management is always looking at new technologies, particularly those being developed by others.

Programming Language Versus Software Environment

Programmers think in terms of languages. Even Visual Studio and Visual C++ are language oriented. This is because, with the exception of VisiSoft, there is no way to build architectures for software. Imagine architects of large buildings being told that their use of engineering drawings is a legacy approach, and is best replaced by writing in a language, e.g., XML, that can draw figures. This is effectively what we have been doing in software. Having used VisiSoft to build large software systems, this is not an absurd comparison. If one has not used this new technology, it is hard to relate to. But having used it, it is hard to imagine not using engineering drawings, and instead, hiding the architecture in code.

But there is much more to a software environment than the drawings and the code. One must keep track of change requests, patches, changed modules headed for the next release, changed modules that did not make the release, test drivers, regression test sets, etc. More importantly, there is a large body of technology that can be applied to all aspects of the software life cycle. The more we can integrate these facilities into an overall environment, the more teamwork becomes important, and the more incentive there is to make that environment better for everyone. This is a management challenge that will be met by good managers.

Hierarchical Software Teams

In 1972, Fred Baker wrote his famous paper on Chief Programmer Teams, [5]. Since Baker worked on the OS-360 project, it became clear that many authors citing his work did not understand the context. One of his points was the limit on the span of control of a single manager. On a project like OS-360, there were a large number of teams, organized hierarchically. These teams were of different types, depending upon what they were doing. Some teams did not actually build software; they built documentation or performed testing. What Baker emphasized were the different skill sets needed to support different teams, and the manageable sizes of the skilled elements.

Borrowing again from the military, thousands of people must be organized to contribute to the cause. One person cannot manage 1000 people. Knowing this, Baker focused on the different types of organizations needed at the bottom layer and the commonality of skill sets required to fulfill the needs of particular types of teams.

Building Large Software Organizations That Are Effective

To control a large software organization, one must first understand the overall software architecture, and how it must be supported with documentation and testing. This leads to an organizational architecture that gets mapped into the software architecture, with specific functions to be performed by each *organizational module*. This implies that one must have a software architecture before one can map out the details of an organization to support it. When building large commercial buildings, the architects are hired first. Once they have developed the drawings and specifications, the job is put out for bid. General contractors pour over the drawings and specifications, and map their subcontractors into the architecture. Why can't we do that with software? (You should have guessed - we do not design architectures!)

In a large organization, higher level managers must deal with lower level managers. Flattening the organization is not nearly as important as having managers who understand what's needed - and what is really being done - below them. If managers do not understand what their subordinates should be doing, and what they are really doing, they cannot manage effectively. This gets worse as an organization is flattened. This points to a major problem in the software industry - finding managers that really understand what is needed and what's going on below. This is because programmers generally like to work on their own and do not aspire to be managers. This is a significant problem to be reckoned with, but one that can be solved through selection and training. This is another management challenge that will be met by good managers.

Policies, Procedures And Standards

American football teams use plays. Every player must learn the playbook. On the field, the quarterback calls the plays, and everyone carries out their assignments from the playbook. If they don't, everyone knows who screwed up. In the military, the generals layout the strategy, and hand it down to the colonels who hand it down to the majors, captains, etc., until it gets down to the sergeants who tell the individual soldiers what to do. When the starting gun is fired, everyone moves out. Neither the football field nor the battlefield has time for bureaucratic decisions. People must make fast decisions using the playbook in a chaotic environment. Programmers generally have much more time. But we still need policies, procedures, and standards, and they must be easily understood and followed. In all cases, managers must have control of the unfolding plan. If someone does not want to play by the book, they must be replaced - immediately! (And we should not have to go through a bureaucracy to remove someone from the team.)

There are many examples of policies, procedures and standards available for software. To highlight the most important factors relative to the new technology paradigm, we offer the following. In general, functional specifications must be produced first. Depending upon the application, this should include a well written user's manual. The functional specifications should be followed by a set of architectural drawings and more detailed software specifications. If R&D efforts are needed, e.g., to see if a particular module can be written to meet a stringent time requirement, then one may write and test some code in the laboratory. Otherwise, the production programmers are not needed until the functional specifications, software architecture and corresponding specifications, e.g., module interfaces, are completed.

Testing of hierarchical modules requires procedures and standards that serve multiple layers in the hierarchy. These must be mapped into the overall software architecture. Testing is an area where procedures and standards must be applied from a practical standpoint. One failure at the bottom of the hierarchy can cause many failures up the chain (as everyone looks for a different bug). Building and managing regression tests must be spelled out clearly so that this important function is performed properly. This is particularly true for utility and library modules.

ARCHITECTURAL IMPACTS

A good architecture is one whose modules are relatively independent. This makes it easier for different people or teams to develop different modules. Thus, assigning the modules to different teams is a critical part of the management process. Modules delineate the boundaries of resources and processes that contain the code. If an architecture is done down to the resource and process level, then the only thing left to be done is to write the code.

In many cases, after coding has started and more is learned about the modules, the architecture must be changed. As in commercial construction projects, architecture affects many aspects of a design, and should not be changed without the review of an architect. So every team must have an architect available to review and make such changes. This provides an independent assessment of such change requests and implementation problems by someone more capable of making the right decision.

In a hierarchical management framework, the architect overseeing the module may require approval from a higher level architect who is concerned with the effects that the module being changed may have on other modules being built by other teams. This should all be apparent from the drawings. Now, imagine that this is all being done without any architecture or drawings. Most of the information needed for these important decisions, including the basic decomposition of the software design, will be hidden in the code.

As a project progresses, the need for utilities and library modules will arise if not provided for in the initial architectural design. In fact, it is common to start combining modules into libraries to minimize testing and to provide well tested modules that can be depended upon when debugging complex algorithms. This can dramatically accelerate the fault isolation process.

The other side of the library picture is that a sufficient number of teams must be assigned to manage the library modules. Experience has shown that, in a highly effective software organization, the number of people building and supporting libraries may be higher than those building tailored application modules. Understanding the importance of investing in libraries requires a long term business management perspective.

LOOKING BACK

Once one has managed the development and support of large software products using the VisiSoft technology, it is hard to envision having it taken away. This is because of the control one has over problem prioritization, resource utilization, and most important the speed with which high quality software is put into a production environment. Being able to look at an architecture from the top, and then being able to drill down to the bottom and look at the code, gives one the ability to find and fix problems very fast. More importantly, one can easily perceive problems before they occur, see how direction must be changed, and determine how those changes may affect the rest of the system. There is no doubt that the real winners with this technology will be the software product managers!

REFERENCES

1. Anselmo, Donald and Henry Ledgard, *Measuring Productivity in The Software Industry*, Communications of the ACM, Vol. 46. No.11, Nov 2003.
2. Anselmo, Donald, *Tools and Software Productivity*, Software Summit, Washington, D.C., May 2004.
3. Anselmo, Donald, *History of the C Programming Language*, Draft, Phoenix, AZ., April 2004.
4. Athans, M. and Falb, P.L., *Optimal Control*, McGraw-Hill, New York, 1966.
5. Baker, F.T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, No 1, 1972.
6. Bach, James, "The Immaturity of CMM," American Programmer, Sept., 1994.
7. Bach, Maurice J., *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
8. Bell, C. G., "Ultracomputers A Teraflop Before Its Time," Communications of the ACM, August 1992, Vol.35, No 8.
9. Booch, Grady, *Software Solutions - Developing the Future*, Communications of the ACM, Vol. 44. No.3, March 2001.
10. Bowers, J.C., and Sedore, S.R., SCEPTRE: A Computer Program for Circuit and Systems Analysis, Prentice Hall, Englewood Cliffs, NJ, 1971.
11. Boehm, Barry, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
12. Brooks, Fred, *The Mythical Man-Month*, Addison-Wesley, Reading, MA, 1975.
13. Brooks, Fred, "No Silver Bullets," IEEE Computer, April 1987.
14. *The Unix Timesharing System*, BSTJ, Vol. 57 No.6, July- August 1978.
15. *The Unix System*, ATT/BTL Technical Journal, Vol. 63 No.8, Oct. 1984.
16. "Is the Computer Business Maturing?", Business Week Cover Story, McGraw-Hill, March 6, 1989.
17. "Can the U.S. Stay Ahead in Software?", Special Software Report, Business Week, McGraw-Hill, March 11, 1991.
18. "PROGNOSIS '95", Business Week, McGraw-Hill, January 9, 1995, pp 72-80.
19. "Software Made Simple", Business Week Cover Story, McGraw-Hill, September 30, 1991, pp 92-100.

20. Wildstrom, Stephen H., "Price Wars Power Up Quality", Business Week, Technology & You, McGraw-Hill, September 18, 1995, pp26.
21. *Industry Outlook*, Business Week, McGraw-Hill, January 12, 2004, pp 92-100.
22. Cave, W., "The Constrained Optimal Design System," Proceedings IEEE WESCON, San Francisco, CA, 1971.
23. Cave, W., and A. Salisbury, "Controlling the Software Life Cycle - The Project Management Task," IEEE Transactions on Software Engineering. Vol SE-4, No 4, July, 1978, pp 326-334.
24. Cave, W., and G. Maymon, *Software Life Cycle Management - The Incremental Method*, Macmillan, New York, NY, 1984.
25. Cave, W., *Software Survivors*, Software Developer & Publisher, W. Cave, July/Aug1996.
26. Cave, W., et.al, " The Effects of Parallel Processing Architectures on Discrete Event Simulation," Proceedings: SPIE Defense & Security Symposium, Mar/Apr 2005, Orlando, FL.
27. Cave, W.C., *Simulation of Complex Systems*, Prediction Systems, Inc., Spring Lake, NJ, June 2001.
28. Camford, Richard, "Software Engineering?", IEEE Spectrum, January, 1995, pp 62-65.
29. Christensen, Clayton M., "The Innovator's Dilemma," Harvard Business School Press, Cambridge, MA, 1997.
30. Constantine, Larry, "Back to the Future," Communications of the ACM, Vol 44, No. 3, March, 2001.
31. Canter, Sheryl, "One-Box Development Systems," Applications Development, PC Magazine, July, 1992.
32. Cusumano, Michael A., "What Road Ahead for Microsoft and Windows?," Communications of the ACM, July 2006, pg 21-23.
33. Daconta, Michael C., *C++ Pointers and Dynamic Memory Management*, John Wiley & Sons, NY, 1995.
34. Deming, W. Edwards, *Out of the Crisis*, MIT CASE, Cambridge, MA, 1992.
35. DeMarco, Tom, *Controlling Software Projects*, Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1982.
36. DeMarco, T and T. Lister, *Peopleware*, Dorset House, New York, NY, 1987.
37. Ferguson, Steve, *The History of Computer Programming Languages*,
http://www.princeton.edu/~ferguson/adw/programming_languages.shtml

38. Fitzsimmons, A., and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys 10, No. 1, March 1978, pp 3-18.
39. Gelb, A., Editor, *Applied Optimal Estimation*, MIT Press, Cambridge, MA, 1974.
40. Gilder, George, *Microcosm*, Simon and Shuster, New York, NY, 1989
41. Gordon, G., *A General Purpose Systems Simulation Program*, Proc. EJCC, Washington, D.C., pp 87-104., MacMillan Publishing Co., New York, 1961.
42. Gordon, G., **The Application of GPSS V to Discrete System Simulation**, Prentice Hall, Englewood Cliffs, NJ, 1961.
43. Gordon, J., *System Simulation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.
44. Groth, R., *Is the Software Industry's Productivity Declining?*, IEEE Software, Nov/Dec 2004.
45. *GSS User's Reference Manual*, Version 10.4, Visual Software International, Spring Lake, NJ, 2005.
46. Guth, Robert A., "Battling Google, Microsoft Changes How It Builds Software," The Wall Street Journal, Sept 23, 2005, page 1, column 5.
47. Hactel, G.D. and Roher, R.A., *Techniques For The Optimal Design And Synthesis Of Switching Circuits*, Proceedings of the IEEE Special Issue on CAD, Nov 1967, pp 1864.
48. Hactel, G.D. et al, *The Sparse Tableau Approach To Network Analysis And Design*, IEEE Transactions on Circuit Theory, Jan 1971, pp 101.
49. Hafner, Eric, *Theory and Design of Oscillators*, Proceedings of the IEEE, New York, NY, in two issues, 1977
50. "DILBERT WOULD RELATE," Managers Bulletin Board, INFOWORLD, February 6, 1995, pp 62.
51. "Calling all COBOL users," INFORM, Digital Equipment Corp., Sept./Oct. 1995, pp 1.
52. Jones, Capers, *Applied Software Measurement: Assuring Productivity and Quality*, McGraw Hill, New York, NY, 1991.
53. Jones, Capers, "Evaluating International Software Productivity Levels," Version 3.0, Software Productivity Research, Inc., Burlington, MA, July, 1991.
54. Juran, Joseph M., "Juran's Quality Handbook," Fifth Ed., McGraw-Hill, 1999.
55. Yasushi Kambayashi and Henry F. Ledgard, "The Separation Principle - A Programming Paradigm" IEEE Software, March/April 2004
56. Kernighan, B.W., and D.M. Ritchie, *The C PROGRAMMING LANGUAGE*, Prentice Hall, Englewood Cliffs, NJ, 1973.

57. Kuhn, Thomas, *The Structure of Scientific Revolutions*, The University of Chicago Press, Chicago, IL, 1970.
58. Dr. Anita J. La Salle, *Software Industry and Economic Security*, Software Industry Workshop, April 27, 2000.
59. Ledgard, H., et al, "The Natural Language of Interactive Systems," CACM No. 10, October 1980, pp 556-563.
60. Henry F. Ledgard, *The Emperor with No Clothes*, Communications of the ACM, Oct 2000.
61. Henry F. Ledgard, *The State of the Software Industry*, Technical Report, University of Toledo, July 2007.
62. Levy, Leon, *Taming the Tiger: Software Engineering and Software Economics*, Springer-Verlag, New York, NY, 1987.
63. Mahoney, Michael S. *The Unix Oral History Project*, <http://www.princeton.edu/~mike/expotape.htm>
64. Maslo, R. and W.C. Cave, A New Approach to Development and Support of Real-Time Control Systems, Proceedings, 7th Annual GSS User's Conference, Prediction Systems, Inc., Spring Lake, NJ, June 1994.
65. Marcotty, Michael, *Software Implementation*, Prentice Hall, New York, NY, 1991.
66. Mills, H.D., Mathematical Foundations of Structured Programming, Technical Report FSC 72-6012, IBM Federal Systems Division, 1972.
67. Mitchell, R., "In Supercomputing, Superconfusion," Business Week, pps 89-90, March, 1993.
68. Netizens: An Anthology, Chap 9, *On the Early History and Impact of Unix - Tools to Build the Tools for a New Millennium*, <http://www.columbia.edu/~rh120/ch106.x09>
69. Parnas, D., "Education for Computer Professionals," IEEE Computer, January 1990, pp 17-22.
70. Patterson, D., Bell, G., et al, "Massively Parallel Uproar," Upside, pps 88-97, March, 1992.
71. Perkins, T., and R. Kalgaard, "Inside Upside," Upside Magazine, September 1991.
72. Pfleeger, S. L., "Viewpoint: Software Engineering Needs to Mature", IEEE Spectrum, January, 1995, pp 64.
73. *A Tale of Three Disciplines and a Revolution*, Jesse H. Poore, IEEE Computer Society, Jan 2004.

74. *Visual Software Development For Parallel Machines*, Final Report, US Army CECOM Contract DAAB07-97-C-H501, Prediction Systems, Inc., Spring Lake, March, 1997.
75. *Multi-Computer Version of GSS*, Final Report, DARPA MHPCC BAA Consortium, Prediction Systems, Inc., Spring Lake, NJ, Sept. 1998.
76. *High Efficiency, Scalable, Parallel Processing*, DARPA Contract SF022-035 Final Report, Prediction Systems, Inc., Spring Lake, NJ, June 2003.
77. Ranum, Marcus J., *SECURITY - The root of the problem*, ACM QUEUE, June 2004.
78. *Development of the C Language*, Second History of Programming Conf., 1993, <http://cm.bell-labs.com/cm/cs/who/dmr/chist.html>
79. VisiSoft General Library - Software Description Document, Visual Software International, Inc., Spring Lake, NJ, March 26, 2011.
80. Rose, F., and R. Turner, "A Jungle Out There", Front Page Article, The Wall Street Journal, January 23, 1995.
81. Rosen, J.P., "What Orientation Should Ada Objects Take?" CACM, Vol 35, No 11, November 1992, pp 71-76.
82. *RTG User's Reference Manual*, Version 4.2, Prediction Systems, Inc., Spring Lake, NJ, 1994.
83. Rubin, K.S., "Reuse in software Engineering: An Object Oriented Perspective," Proceedings of IEEE COMPCON, Spring, 1990.
84. Schweppe, F., *Uncertain Dynamic Systems*, Prentice Hall, Englewood Cliffs, NJ, 1978.
85. Carnegie Mellon University - Software Engineering Institute (SEI) - Capability Maturity Model (CMM) <www.sei.cmu.edu/cmm>.
86. "Chaos, Charting the Seas of Information Technology," The Standish Group International Inc. Report, Dennis, MA, 1995.
87. "Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%," Press Release, The Standish Group International Inc., West Yarmouth, MA, 2003. <http://www.standishgroup.com/press/article.php?id=2>
88. "Standish: Project Success Rates Improved Over 10 Years" Software Magazine, January 2004. <http://www.softwaremag.com/L.cfm?Doc=newsletter/2004-01-15/Standish>
89. *A Description of the Standard File Interface*, Version 2, Prediction Systems, Inc., Spring Lake, NJ, 1995.
90. Sherr, A.L., "Developing and Testing a Large Programming System," In *Program Test Methods*, W.C. Hetzel, Editor, Prentice-Hall, Englewood Cliffs, NJ, 1972.
91. Shore, David, "Computer Technology Stands Poised for Substantial Leap," Signal, February 1993, pp 35-37.

92. Sitner, Jerry, "Viewpoint - How Much Longer," *Mainframe Journal*, July 1990, pp 120.
93. Singleton, R.C., "An Efficient Algorithm for Sorting with Minimum Storage," *CACM*, Vol 12, No 3, March 1969, pp 185-187.
94. Strassmann, Paul, "From a Craft to an Industry," *Ada Symposium*, George Mason University, 1992.
95. Stroustrup, B., "What is Object-Oriented Programming?" *IEEE Software*, May 1988, pp 10-20
96. Stroustrup: What is Object-Oriented Programming? (1991 revised version). Proc. 1st European Software Festival. February, 1991 - public.research.att.com/~bs/whatis.pdf
97. "Musings on the Millennium," Feature Editorial, *Upside Magazine*, October 1994.
98. van der Linden, P, *Expert C Programming - Deep C Secrets*, SunSoft Press - Prentice Hall, Englewood Cliffs, NJ, 1994.
99. *VisiSoft GENERAL Library & RTG_DRAW Library*, Software Description Document, Visual Software International, Spring Lake, NJ, 2006.
100. *VSE User's Reference Manual*, Version 10.4, Visual Software International, Spring Lake, NJ, 2005.
101. Weinberg, G., *An Introduction to General Systems Thinking*, John Wiley & Sons, NY, 1975.
102. Wilbur, M., *Managing Software Reliability: The Paradigmatic Approach*, North Holland, New York, NY, 1981.
103. Yourdon, E, *Decline & Fall of the American Programmer*, Yourdon Press - Prentice Hall, Englewood Cliffs, NJ 1993.
104. Zadeh, L.A. and Desoer, C.A., *Linear System Theory: The State Space Approach*, McGraw-Hill, NY 1963.
105. Zuniga, Gilberto, Concepts for the Implementation of an Air Defense Model Base, Proceedings of the 55th Military Operations Research Symposium, May 1987

Software Engineering

The technology described in this book is a revelation in software. It provides the engineering discipline needed to improve quality, productivity, and run-time speed, while maintaining tight control over extremely large complex software systems. It describes a sound scientific basis for improving these measures. It explains the theory and application of a new approach to building software, particularly when using multiple processors to speed up run times. If you want to know where the software field is headed in the next three decades, read this book. It is the most significant innovation in software since the compiler. Although it takes an engineering background to understand the hard science underlying the concepts, use of the CAD system it describes can be learned at the high school level as well as by subject area experts who want to build their own software.

About the authors

Bill Cave started his career in computers in 1958 at Penn State University, writing computer programs in 1s and 0s on a computer designed and built by the EE department. Receiving the BSEE in 1960, he worked in computer design while getting the MSEE (1963) at night from NYU. Specializing in CAD tools for circuit design, he received a graduate fellowship at Brooklyn Polytech in 1965, and did further graduate studies at Stevens Institute of Technology. Bill started his first CAD company in 1967, providing software for commercial use. In 1974, he founded Prediction Systems, Inc. (PSI) where he remains Chairman of the Board. PSI has an international reputation for building some of the worlds largest simulations and planning tools. In 2004, PSI spun off Visual Software International (VSI), with its tools for building large scale software and simulation systems. These tools are provided to clients as VisiSoft. Bill is an internationally recognized expert in modeling, simulation, and software, having published many articles and books, and lectured in Europe, Australia, and China as well as the U.S. on approaches to building complex software and simulation systems.

Henry Ledgard graduated Magna Cum Laude from Tufts University in 1964. He received the M.S. & Ph.D. degrees in Electrical Engineering from the Massachusetts Institute of Technology in 1969. Starting as a Visiting Fellow at Oxford University in 1969, he went on to the Computer Science department at The Johns Hopkins University for two years, and then to the University of Massachusetts at Amherst for five years. In 1977, he departed from the academic world to be a member of the Honeywell Design Team for the U.S. DoD Common Language Effort (Ada). He followed this with consultancies at commercial companies, including DEC, Alsys, Inc., and Philips Electronics. In 1989, he returned to the academic world as Professor of Electrical Engineering and Computer Science at the University of Toledo, where his specialties are Software Engineering, Object-Oriented Programming, Programming Languages, and Human Computer Interaction. Henry is internationally known as an editor for book companies and journals, as well as a writer of many well known books and articles on software languages.

IETC Publications, Spring Lake, NJ 07762