01 Dec 1993

Process Driven Software Engineering Environments

John Hayes Lampkin

T. Lo

Daniel C. St. Clair
Missouri University of Science and Technology

Process Driven Software Engineering Environments

J. Lampkin*, T. Lo, and D. St. Clair

CSC-93-36

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

*This report is substantially the M. S. thesis of the first author, completed December 1993

ABSTRACT

Software development organizations have begun using Software Engineering Environments (SEEs) with the goal of enhancing the productivity of software developers and improving the quality of software products. The encompassing nature of a SEE means that it is typically very tightly coupled with the way an organization does business. To be most effective, the components of a SEE must be well integrated and the SEE itself must be integrated with the organization.

The challenge of tool integration increases considerably when the components of the environment come from different vendors and support varying degrees of "openness". The challenge of integration with the organization increases in a like manner when the environment must support a variety of different organizations over a long period of time. In addition to these pressures, any SEE must perform well and must "scale" well as the size of the organization changes.

This paper proposes basing the Software Engineering Environment on the software development process used in an organization in order to meet the challenges of integration, performance, and scaling. The goals and services of distributed operating systems and Software Engineering Environments are outlined in order to more clearly define their roles. The motivation for using a well defined software development process is established along with the benefits of basing the Software Engineering Environment on the software development process. Components of a SEE that could effectively support the process and provide integration, performance, and scaling benefits are introduced along with an outline of an Ada program used to model the proposed components. The conclusion provides strong support for process driven SEEs, encourages the expansion of the concept into other "environments," and cautions against literal interpretations of "process integration" that may slow the acceptance of this powerful approach.

TABLE OF CONTENTS

ABSTRACT iii
ACKNOWLEDGEMENTS vi
LIST OF FIGURES vii
LIST OF TABLES viii
SECTION
I. INTRODUCTION 1
A. Background 1
B. The Software Development Process 2
C. Integration 3
D. Performance 4
E. The Operating System 5
F. SEE Components 5
G. Other Environments 5
H. Organization 6
I. Terminology 6
II. OPERATING SYSTEMS 8
A. Classical Operating Systems 8
B. Network Operating Systems 9
C. Distributed Operating Systems 13
D. Distributed Operating System Goals 13
E. Distributed Operating System Services 16
III. SOFTWARE ENGINEERING ENVIRONMENTS 19
A. Tools 19
B. Tool Sets 19
C. Software Engineering Environments 19
D. Software Engineering Environment Goals 21
E. Software Engineering Environment Services 26
IV. COMMON GROUND 31
A. Goals 31
B. Services 32
C. Summary 35
V. SOFTWARE DEVELOPMENT PROCESS 36
A. Motivation 36
B. Definition 38

TABLE OF CONTENTS

VI. PROCESS DRIVEN ENVIRONMENTS 43
VII. BENEFITS OF THE SEE 44
A. The Software Development Process as a Foundation for the SEE 44
B. Team Integration 44
C. Management Integration 45
D. Control Integration 45
E. Performance 46
F. Some Guidelines 47
G. Integration with the Organization 48
VIII. PROCESS COMPONENTS OF THE SEE 50
A. Automating the Software Development Process 50
1. Agents 50
2. Binders 52
3. Scheduler 52
4. Dispatcher 53
B. Performance 53
1. Scenarios 54
2. Analysis 63
C. Scalability 65
IX. MODEL 66
A. Implementation 66
1. Active Components 66
2. Event Trees 67
3. Communication 68
B. Experiments 69
X. CONCLUSIONS 70
REFERENCES 71
VITA 74

ACKNOWLEDGEMENTS

Acknowledgement in this thesis is little recognition for the support provided by my wife over the term of its development. There are some things that can not be fully expressed in words, no matter how carefully they are thought out. It's not easy caring for two small children (no matter how well behaved) while your spouse spends time doing school work. Somehow she kept everything going, and for that, I am eternally grateful.

LIST OF FIGURES

Figures		Page
1.	Software Process Template	3
2.	The ISO OSI Reference Model	10
3.	SEE Framework Reference Model	21
4.	User Interface Development Model	29
5.	Sample OSF / Motif Window	29
6.	The Key Process Areas by Maturity Level	37
7.	Percent of Organizations in Each Process Maturity Level	38
8.	Sample Development Process	41
9.	Placing CASE Tools in the Context of an Organization	49
10.	Major Components Supporting Process Integration	51
11.	Best Case Agent to Agent Communication	54
12.	Performance Scenario 1	57
13.	Performance Scenario 2	58
14	Performance Scenario 3	60
15.	Performance Scenario 4	61
16.	Performance Scenario 5	62
17.	Performance Scenario 6 – Worst Case	63

LIST OF TABLES

Tables		Page
I.	Transparency in a Distributed Operating System	14
II.	Service Terminology	33
III.	Scenario Overview	56
IV.	Scenario Summary	64

I. INTRODUCTION

The broad objective of this research was an improved method of developing software. Researching this general topic led to the more focused objective of establishing a mechanism to integrate the software development process with the Software Engineering Environment (SEE). Requirements of integration were that it should be done in such a way that it provided context for tool communication; context for user interaction; and support and enforcement of the software development process. The resulting system should be capable of handling changes to the software development process, the software, or the hardware that comprise the SEE. The system should also perform well in a distributed environment.

As this paper documents, the objective was met by defining a collection of active SEE components. The components integrate the software development process with the SEE and they place all SEE activity within the context of the software development process. New components can be added readily. Existing ones can be modified or removed as necessary. The components communicate in a manner that allows their addresses to be determined at run time. An additional benefit to the approach is that it is not exclusively tied to the software development domain.

A. BACKGROUND

Since the invention of the first computer, the world has become increasingly dependant upon computer systems. In order to meet the rising demand with a quality product, software development organizations have increasingly defined their own Software Engineering Environments (SEE) to assist software development. Initial SEEs simply provided a collection of "tools" to automate some of the tasks involved in software development. Today, the emphasis is on integrating the tools in the SEE to provide software development teams with a coordinated, highly productive environment.

With many software development organizations pursuing the same goal, a new industry has emerged over the last decade whose purpose is to develop and sell SEEs (or portions thereof). Naturally, this industry is addressing the problem of integrating tools in the SEE. Some companies have addressed the problem in a "closed" fashion – one in which they do not make the key elements of their tools or data easily accessible to other tools in the environment. Others have addressed it in an "open" fashion – where the company makes key elements of their tools and data easily accessible to other tools in the environment. Typically, the closed solution has been the approach of companies

that intend to provide a total SEE solution. Companies addressing a portion of the solution tend to approach it in an open manner.

While economic arguments can be presented for both approaches, the open approach appears to have the edge. Witness, for example, Rational Corporation's evolution of the Rational Environment™ from a proprietary system that ran on a proprietary hardware platform to Rational Apex™, an environment that runs on Unix platforms [1]. Witness also the SEE provided by Digital Equipment Corporation. Digital's SEE is large and attempts to provide a total solution. None-the-less, it does so in an open manner, through standards [2]. Many other cases could be cited with just a quick look at the industry, but the point here is not to argue for one approach over the other, it is to observe that the more flexible approach is winning the battle of supply and demand.

It is natural for a flexible approach to be preferred over the long term. Tools that are the best in class today may be the worst tomorrow. A SEE that lets an organization unplug the old tool and plug in the new tool will carry a distinct advantage. There is another issue, however. No software organization is like any other. Each organization has its own standards and procedures. It is important for a SEE to allow the organization to define its own method of developing software – its own process.

B. THE SOFTWARE DEVELOPMENT PROCESS

A substantial amount of work has been done in the area of software process definition by the Software Engineering Institute (SEI) at Carnegie Mellon University. The SEI has created a Capability Maturity Model (CMM) and an assessment and evaluation methodology that allow the process maturity of an organization to be rated [3]. With the emergence of the CMM, software organizations have increased their emphasis on well defined and documented software development processes. Figure 1 shows an example of the type of diagram typically used to show the tasks (the rectangles in the diagram), task relationships, and participating roles for a process. Typically, a short task description would be provided in the task rectangles shown in Figure 1. Other pages would then be used as required to provide detailed information such as the entry criteria and exit criteria for individual tasks, a more complete task description, and a list of products.

The software development process is designed to ensure that all checks and balances are in place in order to provide a quality product every time. The process has a heavy influence on the culture in an organization. A SEE that supports or assists the process will be more readily accepted by the developers than one that does not.

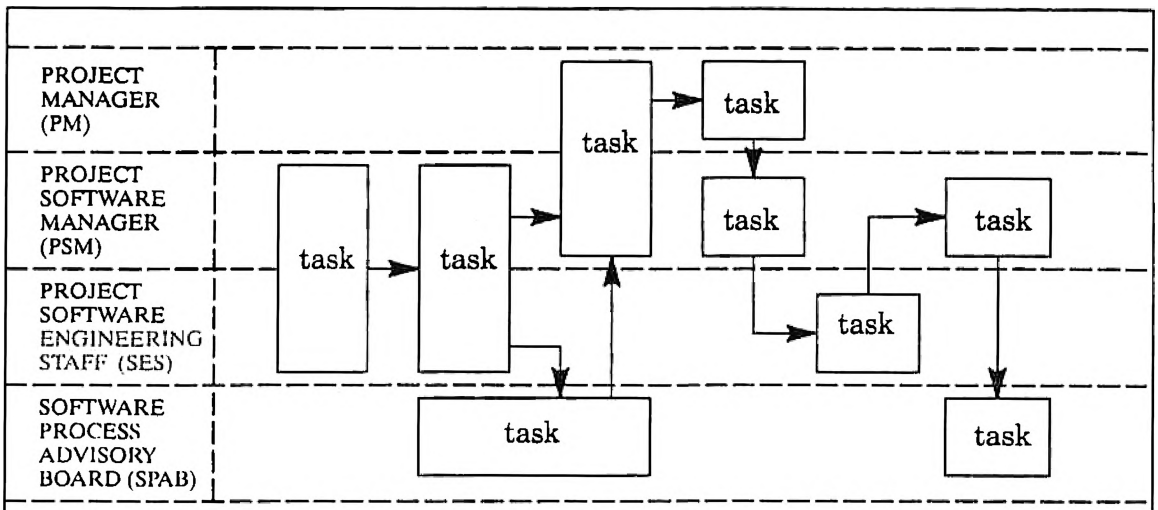

Figure 1. Software Process Template

C. INTEGRATION

It is somewhat enlightening, then, to observe that through the short history of SEEs, the kinds and degrees of integration have typically focused on the technical nature of the environment [4, 5, 6]. Questions such as the following were used to determine the degree of integration:

- Do the tools provide a consistent user interface (presentation integration)?
- Can the tools use the same data without transformation (data integration)?
- Can the tools use services provided by one another (control integration)?

Software Engineering Environment integration was evaluated purely in the context of the components within the SEE. Integration of the SEE within the context of the organization was not considered. The focus has shifted in the last few years.

Articles have started to appear that discuss SEE integration within the context of the software development organization [4, 7, 8]. It is becoming more generally recognized that an environment that offers familiar paradigms to the software developer is more readily accepted, used, and effective than one that does not. Assuming a competent user interface, a process driven environment is familiar and almost intuitive because it simply supports the process that developers already use.

Organizations that develop SEEs are more encouraged as they begin to understand the potential productivity gains that a process driven approach is able to offer. Additional encouragement has been provided by recent advances in the areas of presentation, data, and control integration. Witness, for example, the number of

products that have been built upon the Field model of coarse grained control integration [9], included are: HP SoftBench, HP CASEdge, SUN ToolTalk, DEC FUSE, and IBM SDE/6000 [10]. In addition to supporting control integration, these products all share another important characteristic – network transparency.

Integrating the tools that comprise a SEE typically requires integrating tools that run on different hardware platforms, possibly running different operating systems. For example, a developer on a VAX VMS workstation might be running a compiler and linker locally, a testing program on a Silicon Graphics Unix machine, and a configuration management program on an HP Unix machine. An integrated SEE might allow the developer to select an icon representing a particular source unit from the configuration management system and drag and drop the source unit icon onto the icon representing the compiler. The source unit would automatically be checked out of the configuration management system and compiled. A more sophisticated SEE might allow the source file to be dropped onto an icon representing the linker. The linker tool would determine whether this unit, or any units upon which this unit depends, require compilation. If compilation is required, the files would be automatically checked out of the configuration management system, compiled in the proper order, and then linked to form an executable file. This scenario can likewise be extended to run the appropriate test cases after building the executable. Further extensions can be made to include metrics generation and collection, documentation updates, and so on. The point is that the components of the SEE work together locally or across a network and they do so in a manner that is transparent to the user.

Performance issues become important quickly. Sending a message to a tool on a remote machine takes a lot longer than making a subroutine call. The more interaction between tools that is required, the bigger the potential performance problem. To exacerbate this situation, data and control integration are reaching lower and lower levels. Data integration at the file level is known as "coarse" data integration. "Fine" data integration deals with items such as individual source statements in a program or individual paragraphs in a document. Control integration at the tool level is known as "coarse" control integration. "Fine" control integration deals with the function or procedure level. The trend is toward fine data and control integration [10]. Supporting fine levels of integration increases the overhead required for tool cooperation and thus increases the performance demands on the SEE.

D. PERFORMANCE

An approach to providing both fine and coarse grain data and control integration has been outlined by Harrison, Osher, and Kavianpour [10]. They call their approach

"Object-Oriented Tool Integration Services (OOTIS)". Their approach is based upon object oriented database technology and focuses on the performance demands of fine grained data and control integration. While this technology is necessary and important, a weakness seems to remain in one area of coarse grained control integration. When one tool requires the services of another tool and the second tool is not currently available on the system, the first tool must wait for the second to become available before it can proceed. Examples would be logging into a database, or loading a tool into memory for execution. On some systems, this could cause a considerable delay.

Later sections of this paper will show that a process driven SEE can relieve this problem. A process driven SEE does not remove the need for performance improving technologies such as those used by OOTIS, however, and neither of these solutions removes the dependence of the SEE upon the services provided by the operating system. Whether integration is coarse or fine, some bindings will be made at run time because the system must be assumed to be dynamic. A SEE component can not always expect another component with which it communicates to be on the same machine in the network every time. It may move dynamically based upon load, it may move simply because the system administrator moved it, or it may not be there at all because the machine is down. In any case, no assumptions should be made and it is likely that operating system services will be used.

E. THE OPERATING SYSTEM

A SEE provides the environment in which software developers work, but it does not replace the operating system. A look at the goals of and the services provided by both operating systems and SEEs reveals that the two seem to be converging. Topics such as distributed computing are being addressed in both areas. It seems likely that some of the functions being built into todays SEEs will be in tomorrows operating systems.

F. SEE COMPONENTS

The trend toward distributed computing will have a heavy influence on the way components of a SEE are integrated. In fact, it can seem to stand at odds to the notion of integration. Both integration and distribution are essential features of SEE components, however. SEE components must work together across the network to accomplish the objectives of the SEE. A set of components are introduced in this paper that combine to provide process integration in a distributed manner.

G. OTHER ENVIRONMENTS

It is important to note that while this paper discusses process driven Software Engineering Environments, the concept of process based computing can and should be

applied to other areas. The concept can be applied to any area where it is helpful for the software in use to have an understanding of the context in which the user is working, or where there is a desire for a number of software components to work together to assist the user.

Would it be beneficial, for example, for a home PC to automatically disable the children's software when it was bed time; for it to prevent them from changing the clock setting (i.e. bed time); for it to automatically load a spreadsheet program and the appropriate file when mom or dad selected 'Balance the Checking Account' from a menu? These examples may stretch the point, but they demonstrate the application of this concept in what may be a more familiar domain. The computer was provided with certain rules of the house and simply enforced them. Properly written rules will allow the freedom of experimentation because the rules will prevent harmful situations like data loss. In a household, this may not be a large problem, but in organizations where 200 people may be working on a project at one time, there is always someone who did not get the word. The ability to reach those people through the process in real time is a powerful tool.

H. ORGANIZATION

This paper is divided into ten major sections. Section I begins the paper by providing the objective and exploring the general topic. Section X provides a summary of the conclusions.

Sections II through IX can be divided into two parts. The first part, sections II through IV, provides a review of the literature on the subjects of operating systems and Software Engineering Environments. It builds evidence that operating system services and SEE services appear to have some overlap. The second part of the paper, sections V through IX provides a review of the literature on the subjects of software development processes and process driven Software Engineering Environments. It builds evidence to support the use of the software development process as a fundamental element in a SEE. It also introduces SEE components designed specifically to provide process integration that supports tool integration, performance, and scaling.

I. TERMINOLOGY

The word "process" represents a particular challenge when the operating system and software development process domains are discussed concurrently. In general, the context should make the meaning of the word sufficiently clear throughout this paper. However, in some cases, the term "software development process" is used in place of "process" to ensure clarity.

The term Software Engineering Environment (SEE) is not universal. The term Computer Aided Software Engineering (CASE) is often used for similar purposes. Where CASE or iCASE (integrated CASE) were used in material referenced by this paper, they were not modified and will thus appear.

II. OPERATING SYSTEMS

Operating systems perform two basically unrelated functions. They provide users with a convenient interface and they manage the system resources [11]. Operating systems can be placed into categories based upon their characteristics. Though the names differ slightly among references, there are generally three categories of operating systems: classical, network, and distributed.

The goal of this section is to introduce distributed operating systems. Classical and network operating systems will be introduced first in order to provide some background.

A. CLASSICAL OPERATING SYSTEMS

The classical operating system provides resource management and an interface that reduces the effort required to manage hardware resources. A classical operating system may run on a computer with multiple processors if the processors share memory.

Services of a classical operating system can typically be grouped into the following five major categories: Command Interpretation, Processes, Memory Management, File Systems, Input / Output. To reinforce the close tie between an operating system and the hardware, it is worth noting that the last four categories map directly to the following major hardware components: CPU, Memory, Secondary Storage, Input / Output Peripherals.

Commands can be provided directly by a user via some input device such as a keyboard or mouse, or they can be provided by a user's program. Each command must follow a specified syntax in order to be recognized. Once a command is recognized, the operating system attempts to execute it.

A process is a program in execution [12]. System users initiate the execution of processes using operating system commands. On a multiprogramming system, when one process is performing an operation that does not require the CPU (e.g. I/O), that process gives up the CPU so that a process that is ready to use the CPU may do so. On a multiprocessing system, many CPUs are available to execute processes at the same time. With regard to processes, the operating system will normally:

- Create and delete processes
- Suspend and resume processes
- Provide for process synchronization

- Provide for process communication
- Handle deadlocks
- Schedule processes for execution

A file is a logical storage unit on the system. The physical properties of a file are defined by the particular storage device (e.g. disk, tape). Files are typically organized into directories. Directories provide hierarchical organization of files. With regard to files, the operating system will normally:

- Map the logical file onto the physical device
- Provide consistent file access regardless of device
- Control file access
- Create and delete files
- Create and delete directories
- Support primitives for manipulating files and directories
- Provide protection from unauthorized access

Memory is an array of bytes, each with its own address. A process must be placed in memory in order for it to be executed. A file must be placed in memory in order for it to be accessed. With regard to memory management, the operating system will normally:

- Decide which files and processes are to be loaded into memory when memory space becomes available
- Keep track of which parts of memory are currently being used and by whom
- Allocate and deallocate memory space as needed

Input to a computer can be received from devices such as keyboards and disk drives. Output provided by a computer can be received by devices such as terminals and disk drives. With regard to input / output, the operating system will normally:

- Cache disk accesses in memory
- Communicate with device drivers
- Schedule disk accesses
- Allocate storage
- Manage free space

B. NETWORK OPERATING SYSTEMS

As the number of computers increased over time, so did the desire to connect machines to one another. Network operating systems were introduced. Network

operating systems provide the same functions as classical operating systems while providing the additional capability to access files and other computers across the network. Each computer runs its own network operating system.

The difference between a classical operating system and a network operating system is not large. The network operating system requires the addition of a network interface controller and software to drive it. In addition, it requires software to perform remote login and remote file access. The additions are most effectively described with the use of the International Standards Organization (ISO) [13] Open Systems Interconnection (OSI) Reference Model [14, 15], shown in Figure 2.

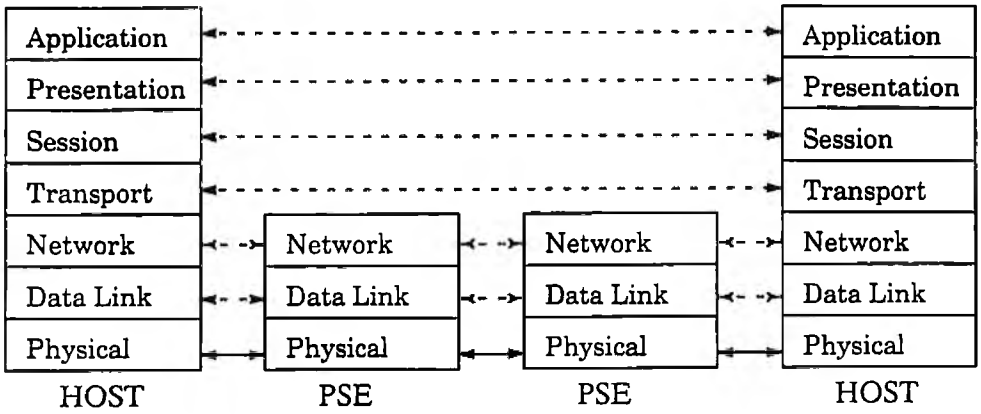

Figure 2. The ISO OSI Reference Model [16]

There are seven layers in the ISO OSI Reference Model. Every layer in the model defines a different set of software capabilities that distinguish a network operating system from a classical operating system. The lower three layers primarily provide data communication functions while the upper three layers primarily provide data processing functions.

Logically (see dashed lines in Figure 2), each layer communicates with the corresponding layer on a different machine in the network. In reality, however, only the physical layers have direct connection. The other layers communicate directly only with the layer above and the layer below. Data from the application layer is handed to the presentation layer which performs some transformations and passes it to the session layer, etc. Once at the physical layer, the data is moved from one machine to another. In a local area network, data is typically broadcast to all machines. Each machine will only accept data addressed to it. In a wide area network, Packet Switching Exchanges (PSEs) are likely to be involved [16]. A PSE is simply a special purpose computer that sends and receives packets of data through the network on behalf of other computers.

Either way, once the data has reached the targeted machine, it moves from the physical layer up to the application layer, with each layer processing and removing overhead information added by the corresponding layer on the sending machine.

Logically, a collection of data passed from one machine to another is called a message. In order for the definition of a message to be generic, however, a message must be allowed to be any length. Since the physical buffer used by the hardware that provides data communication must be a fixed size, messages are typically further divided into packets. A length restriction is applied to packets.

Predefined protocols, which are precisely-defined rules, are used to ensure proper communication between a layer on the sending machine and the corresponding layer on the destination machine. The following paragraphs describe each layer [16, 17].

The application layer consists of protocols with which an application gains access to the network. For each application class that requires network communication, an application level protocol is required. For two application processes to exchange meaningful information, they must agree on the semantics of all aspects concerning the intended exchange of information. Examples of application level protocols include terminal emulation protocols and file transfer protocols.

The presentation layer provides data format conversions to place the data in an external data representation before it is transmitted across the network. The presentation layer also provides data format conversions to read the external data representation and convert it to a format that is known to the machine receiving the data. The following cases provide good examples of the usefulness of the presentation layer:

- When the bit order in machine words on the source (e.g. an IBM) are in a different order from those on the destination (e.g. a VAX)
- When data is encrypted for additional security while on the network
- When data is compressed for improved performance

The session layer establishes and maintains a virtual connection called a session between processes in different machines connected by a network. A process on one machine works through the application and presentation layers on the same machine to tell the session layer the name of a service that it wants to use on another machine. The session layer then works with the session layer on the destination machine to establish a virtual connection with a server process on the destination machine. One of the processes then makes the virtual connection name known. If the other process requests a connection with the same name, a virtual connection has been established

between the two processes. Subsequent communication between the processes takes place through this virtual connection. The connection can be terminated by either process.

The transport layer provides network independent message transport service between machines on a network. The following services are typical of the transport layer:

- Translates network independent transport addresses into network specific addresses
- Segments messages into appropriately sized packets at the source, and reassembles them in the proper sequence at the destination
- Ensures that messages are transferred reliably

The network layer transfers data packets across the network. It relieves the transport layer of the need to know anything about the operational characteristics of the specific transmission facility.

In a wide area network, the network layer would establish a route from source to destination, through as many intermediate nodes as required. In a local area network, the network layer functions are usually handled by the data link layer. When this is the case, the network layer is not required.

The data link layer is responsible for the error-free transmission of packets across a network. The following services are typical of the data link layer:

- Initialization.
  Initialize a link.
- Mechanism to Segment Information.
  Subdivides data packets into blocks (or frames) in order to increase the chance of transmitting without error in a noisy environment.
- Error Checking.
  Errors are detected, and where possible, corrected.
- Data Synchronization.
  The receiver must align a character decoding mechanism to match the character encoding mechanism of the sender.
- Flow Control.
  Ensure that a sender does not transmit an amount of data that exceeds the receiver's ability to handle it.
- Abnormal Condition Recovery.
  Detect and recover from lost connections.

- Termination.
  Terminate the link (hardware is still connected).

The function of the physical layer is to transmit data bits over some medium connecting two pieces of communications equipment. This layer includes the hardware that drives the network and the circuits themselves.

Most equipment today is analog, requiring amplitude or frequency modulation to transmit data. The trend toward digital communication equipment will reduce the complexity of this layer in the future.

C. DISTRIBUTED OPERATING SYSTEMS

As hardware costs have continued to decrease while performance has increased, more powerful software applications have become available. Users again have the desire to increase their capability to interact. Enter the concept of distributed operating systems. A distributed operating system provides network resource management and an interface that reduces the effort required to use network hardware resources. It provides the same services as the classical operating system except the services apply across a network of computers that do not share memory. Logically speaking, the network ends up looking like one big computer to the distributed operating system.

The distributed operating system has to walk a line between separation and transparency. The physical distance among computers creates difficult problems to solve when the objective is to create the appearance that all computers are acting as one.

In addition to the typical services provided by classical and network operating systems, a distributed operating system will add the services described in section II.E.

D. DISTRIBUTED OPERATING SYSTEM GOALS

The primary goal of a Distributed Operating System is to provide a single system view of the entire network. This view must be achieved at both the system user and system programmer levels. The following paragraphs address the goals of transparency, performance, reliability, and scalability.

Unless it affects performance, or the method of access, system users are usually not concerned that a particular program is running on a specific computer or that a particular file is on a specific disk. Programs, on the other hand, may be written to interface with other programs using mechanisms that are highly dependant upon the

computer on which they are running (e.g. a global section of memory). In short, the single system view is easier to provide to a user than it is to a program.

Tanenbaum [11] defines five types of transparency that a distributed operating system should provide. These five types are shown in table I.

Table I. Transparency in a Distributed Operating System [11]

Kind	Meaning
Location transparency	Users and programs can not tell where the resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	Users and programs can not tell how many copies exist
Concurrency transparency	Multiple users and programs can share resources automatically
Parallelism transparency	Activities can happen in parallel without users or programs knowing

Location transparency allows resources to be accessed in a manner that is totally independent of their location. The name of a resource should not be tied to its location.

Migration transparency allows the system to be reconfigured or files or databases to be moved without affecting the way the resource is accessed. This is similar to location transparency. The difference is that migration transparency not only says that it does not matter where a resource is, the resource may also be moved.

Replication transparency allows the operating system to make multiple copies of a widely read file, for example, at various locations across the system. This can improve performance by changing many long-distance network references into local cluster or possibly computer references.

Concurrency transparency allows the operating system to handle a case when, for example, more than one user tries to write to the same file at the same time without corrupting the file and without resorting to an error message to the user. The operating system will automatically synchronize access to resources accessed by more than one user.

Parallelism transparency allows a programmer, for example, to ignore the fact that their system includes a certain number of CPUs that can be used to solve a highly parallel problem. Instead of describing how to break up a task among various processors, the programmer can count on the operating system to break it up appropriately.

An advantage of a distributed system is that the workload can be distributed across a number of computers in a parallel fashion. While this naturally speeds up processes that are relatively autonomous, when processes communicate frequently, the additional network traffic may present a bottleneck that slows the overall performance beyond a non-distributed solution. The decision to migrate a process to a remote host has both a direct (processor speed) and indirect (network traffic) affect on performance. Where processes exhibit very little interprocess communication, processor speed is a more important factor. Where processes exhibit a lot of interprocess communication, network traffic is a more important factor. A system that automatically distributes processes should be able to dynamically adjust for these factors.

That being said, any system that automatically distributes processes across a geographically dispersed area must also consider the geographic distance from the computer where the process request is being made. As a rule, the greater the distance the longer the propagation delay. Even a process that has little communication with other processes may be unacceptably slow if the distance between communicating processes is too great. In addition, the time to simply load a process into the memory of a remote computer in order for it to execute may be prohibitive.

While these considerations are important, it is also important for a distributed system to acknowledge that different jobs have different priorities. Some can tolerate slow performance while others can not.

The "grain size" of parallelism is typically a primary factor in performance. Fine grain parallelism might execute individual source code instructions in parallel, although distributing them across a network would have questionable value. Coarse grained parallelism might execute entire programs in parallel.

Leslie Lamport [11] has described a distributed system as, "One on which I cannot get any work done because some machine I have never heard of has crashed." The goal of a distributed operating system should not only be performance improvement through the effective distribution of processes across network components, but also the reliable execution of all tasks. Where a failed task can only be tolerated in extreme conditions, fault tolerance may be built into the system to allow recovery or at least to maintain data integrity.

A distributed operating system may not notify the user of errors that occur during processing if it can recover and complete the task. This is analogous to a tape drive that encounters an error reading a tape but uses an error recovery routine to derive the data and thus recover from the error.

Deitel [18] has outlined techniques commonly used to recover from equipment failures they are:

- Multiple copies of critical data for the system and the various processes should be maintained.
- The operating system is designed to run under degraded conditions.
- The hardware includes error detection and correction capabilities.
- Idle processor capacity is used to perform search for potential failures before they occur.
- The operating system directs a functioning processor to take control of a process running on a failing processor.

It is important that the system does not assume a minimum or maximum number of resources. Future systems will likely contain many more processors and be more geographically dispersed. Witness the French Post, Telephone and Telegraph administration which is installing a terminal in every household and business in France [11]. These terminals will form a huge country-wide network.

It is also possible, however, for a system to downsize. To remove unwanted traffic or communication, a cluster of computers may be removed from a larger network. The operating system should be able to handle this.

E. DISTRIBUTED OPERATING SYSTEM SERVICES

The following paragraphs describe services provided by distributed operating systems in order to meet the goals outlined above. On the whole, the services described below focus on single, global mechanisms that perform the same operation regardless of their location and regardless of the location of the user.

A single, global interprocess communication mechanism should be provided. Processes will be distributed to different hosts across the network depending upon current load conditions. When two processes communicate, they must be allowed to communicate in the same manner regardless of the hosts on which they are running. A global mechanism provides a level of transparency that allows processes to run correctly without change and without being conscious of their host machine or the host machine of the process with which they are communicating.

A single, global protection scheme should be provided. Files and, in some cases, commands can be protected from access by certain users. A single, global protection scheme allows the operating system to protect against unauthorized access in a

consistent way with minimal overhead. It also provides a level of transparency to processes or users that wish to modify access control. Regardless of their host machine, they use the same commands.

A single method of process management should be provided. Section II.A lists process management functions that an operating system typically performs. Without a single method of process management, deadlock detection and process scheduling are very difficult to perform. In addition, it is possible that different methods of process management could lead to different results depending upon the hosts that happen to be chosen. Providing a single method of managing processes across the network allows users to understand how their processes will be managed and it ensures that processes will be managed in the same way every time they are run, regardless of the hosts on which they execute. User and system processes that monitor process execution and take action based upon results will also be far easier to maintain.

As an example of the difficult issues related to process management, the following case is offered. The designer of a distributed system would want to allow one program to be divided into parts that could be run on several processors around the network in parallel. All of this must be done in a manner that is transparent both to the program's author and to its user. While this concept can be stated and understood in these simple terms, the solution will be very complex. Among other things, the distributed operating system will have to schedule many different parts of many different programs across many different computers. Each computer may have a different effective processing speed. The overall network performance must be maintained or improved through it all, else there was little reason to provide the distributed operating system.

A single set of system calls should be provided. Processes that make calls to operating system routines must have a single set of calls from which to select. In addition, these calls must make sense in a distributed environment. In other words, the system calls used by a process should not limit its use to particular host computers on the network, and the system calls should not be restricted to those present in an operating system that supports a single computer.

Operating system kernels should be identical on all CPUs. With a single set of system calls and required coordination of global activities, identical operating system kernels would increase the reliability of the system and reduce the maintenance costs. Each kernel could manage the local memory and schedule local processes.

A single, global file system should be used. If a process can execute on any host in the network, it is essential that file access be consistent regardless of the host. A single, global file system will provide this. In addition, performance gains can be made by

migrating a file around the network depending upon the location of the process that is currently accessing the file. With the potential that a file has been relocated (migrated) by the operating system, and the potential that a number of different users may want to write to one file, a single, global file system is a natural choice.

III. SOFTWARE ENGINEERING ENVIRONMENTS

In the following paragraphs, the evolution of the Software Engineering Environment is explored. In reading the paragraphs below, it is useful to think of the development of software as a series of phases on a horizontal plane. This is intended to be generic and is not meant to propose a waterfall, spiral, or any other approach to software development.

A. TOOLS

A "tool" is an application program that provides assistance in performing some task. Early environments for software development were comprised of individual tools that either performed a basic function on many objects (e.g. an editor) or performed a very specific function on specific objects (e.g. an assembler). From the perspective of a software life-cycle with phases on the horizontal plane, there was little or no horizontal integration among tools.

B. TOOL SETS

In an effort to improve software developer productivity, horizontal integration among tools began to take place. In some cases, the software developers initiated the integration themselves [5]. Tool sets began to emerge that provided horizontal integration among tools within a single life-cycle phase, or in some cases among multiple life-cycle phases. Typically, tool sets that span more than one phase work with objects in adjacent phases. In no case does a tool set provide all operations required on all objects across the entire life-cycle.

C. SOFTWARE ENGINEERING ENVIRONMENTS

A Software Engineering Environment (SEE) has been defined as "a software based system which provides automated support for the engineering of software systems and for the management of the software process [19]." It is an attempt to achieve horizontal integration across all life-cycle phases. Because different vendors typically provide products that support individual or adjacent phases, implementation of a SEE has proven elusive.

The motivation for capable Software Engineering Environments has primarily been economic. Paul Strassman has compared current software development to the

medieval guild environment "... where each town makes its own shoes, kills its own cows, tans its own leather and, if you're lucky, three years later you get custom-made shoes that cost a great deal of money [19]." Mr. Strassman is the Director of Defense Information. He goes on to say, "The number one priority of the Department of Defense, as I see it, is to convert its software technology capability from a cottage industry to a modern industrial method of production [19]."

With this thought in mind, it is interesting to observe that Norman and Chen anticipate "The field of software metrics will continue to grow in the 1990's, because using metrics in conjunction with tools and methods and applying statistical process control will help us better manage the development process [20]." This statement would seem to support Strassman's number one priority, as it is directly in line with the proven approach used by Edward Deming to improve quality in manufacturing industries. Deming is a proponent of defining, measuring, controlling, and engineering the characteristics of processes and products.

Software Engineering Environments today do little to support this philosophy, but they are working toward it. A current SEE is typically a collection of "integrated" application programs (software tools), each of which functions in a fairly autonomous manner. Integration among tools has been achieved to varying degrees and will be discussed below. The Reference Model for the Framework of a Software Engineering Environment, accepted by the National Institute for Standards and Technology (NIST), formerly the National Bureau of Standards, and the European Computer Manufacturers Association (ECMA) [21], will be used to aid the discussion.

A SEE framework, however, is more than just a model. Framework products are being developed whose sole function is tool integration. The SEE framework can be defined as "The infrastructure for tool integration. A product whose main role is to integrate a set of Computer Aided Software Engineering tools while providing little direct functionality of its own [19]." The products listed in the second paragraph in section I.C. (HP SoftBench, ...) would all be considered frameworks.

The NIST / ECMA model was developed with three objectives in mind [19]:

- To support the identification of areas in SEE architectures requiring further development
- To provide a common reference for describing existing standards
- To improve standards

The reference model is shown in Figure 3. Because of appearance, it is often called the "toaster model." The SEE framework products provide the services identified by

the shaded areas in Figure 3. The model logically divides services provided by the framework. The services are discussed later.

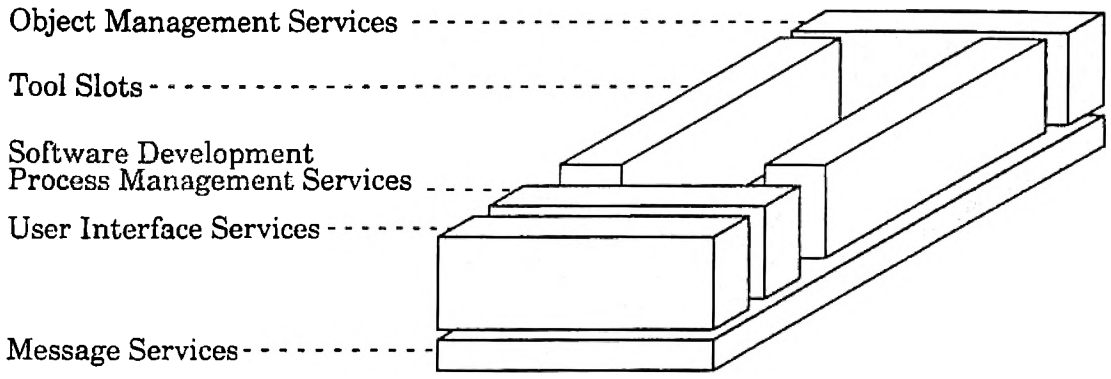

Figure 3. SEE Framework Reference Model [19]

D. SOFTWARE ENGINEERING ENVIRONMENT GOALS

The goal of a Software Engineering Environment is to "support software development – specifically, to provide a software-engineering team with a productive and efficient environment so that engineers can produce high-quality software on time and within budget [6]." It is interesting to note, however, that SEE developers usually discuss their goals in terms of the two competing notions of open systems and integrated systems.

An open system is one in which the steps required for an application to be considered part of the system are both easy and well defined. It is important for a SEE to be open because it is widely recognized that tools from various vendors will be combined to form the full SEE. As new tools become available, possibly from different vendors, they should be incorporated into the environment if they improve the SEEs ability to support software development.

An integrated system is one in which all of it's "components function as part of a single, consistent, coherent whole [22]." This definition brings to mind the distributed operating system goal of a single view of the system. In the case of the SEE, the components are limited to software components, but like the distributed operating system, integration must be provided at both the system user and system programmer levels.

One way to meet both open and integrated demands is through the use of interface standards. With well defined interface standards, tools can be plugged into an

environment effortlessly, and they will immediately be integrated with the other tools in the environment. Currently, there is disagreement about where standards should be applied. Never-the-less, everyone seems to agree that the answer lies in a better understanding of integration.

Much of the research to this point in the area of integration has used the definitions supplied by Anthony Wasserman [23]. As discussed before, however, a trend toward characterizing the integration of a SEE with the organization in which it is used has begun. This is an important trend, but should not be followed to the exclusion of the previous work. Both are and will remain relevant. Brown and McDermid would argue this point, saying that the only integration that is relevant is that with the organization [6].

Looking at the two camps in more detail, Anthony Wasserman has identified five kinds of integration: platform, presentation, data, control, and software development process [23]. (Many researchers use only the last four to define the scope of their analysis related to a SEE.) Brown and McDermid argue that this approach loses sight of the real purpose of a SEE. They too have identified five kinds of integration: interface, software development process, tool, team, and management [6].

Comparing the approaches, it can be seen that "software development process" integration is identified in both. Brown and McDermid's "interface" integration is equivalent to Wasserman's "presentation" integration. Wasserman's "data" and "control" integration can be mapped into Brown and McDermid's "tool" integration. "Team" and "management" integration remain a focal point of Brown and McDermid but were not included in Wasserman's analysis. The following paragraphs will describe these categories in more detail.

The goal of presentation integration is to improve the efficiency and effectiveness of the user's interaction with the environment by reducing his cognitive load [22]. Since a SEE is typically comprised of a number of independent tools, each tool may have its own user interface. The user interface for each tool may use different commands, key stroke sequences, or mouse sequences to accomplish equivalent functions. When the same person uses both tools, that is confusing. A highly integrated SEE would have few, if any, of these differences in the user interface. The cognitive load can be further reduced if the interface is intuitive.

The two major properties of presentation integration are 1) appearance and behavior, and 2) the interaction paradigm. These are explained below.

Tools that use the same symbols for the same purpose; that use the same verbs and commands for the same operations; and, that exhibit equivalent response times could

be said to have appearance and behavior integration. The most important result is that a user can move from one tool to the next without changing their mental model of the system.

Some tools present graphical objects to their user in order to promote a mental image of the system. For example, a data file on the computer may be accessed by first opening an image of a cabinet, then opening an image of a folder, then finally, opening an image of a document. From this presentation, it might be assumed that the user would metaphorically adopt a mental model of the computer file structure that is equivalent to paper filing cabinets. Now suppose that the information in the file was really held in a database and was presented in the file through some document link technology. If the user is asked to update the data in the database directly, unless the same cabinet / folder / document paradigm is obvious, it will be a very difficult job at best. Tools that use the same mental models (paradigms) are said to have interaction paradigm integration.

The goal of data integration is to ensure that all the information in the environment is managed as a consistent whole, regardless of how parts of it are operated on and transformed [22]. Data integration is not relevant among tools that do not share persistent or non-persistent data.

Interoperability is a measure of the amount of work required to allow tools to use the same data. If two tools use the same data but a format conversion must be performed in order for one of the two tools to use it, the tools do not exhibit interoperability integration. Interoperability can be measured from both the user perspective and from the developer perspective. When a user has to take some action in order to allow two tools to use the same data, the tools do not have interoperability integration from the user perspective. When the developer has to take some action in order to allow two tools to use the same data, the tools do not have interoperability integration from the developer perspective.

Tools that use the same data should not maintain duplicate copies of that data. There should be one instance of all data in the system in order to optimize disk space and ensure consistency. This does not preclude replicated data that may be intentionally maintained by a distributed database or operating system.

Under the condition that two data values in a database may have some restrictions relative to one another, it is possible that two tools operating independently on individual data values may violate the value restrictions. Tools that have data consistency integration do not violate such rules.

Data exchange is a form of integration similar to interoperability. Where interoperability applies only to persistent data, data exchange applies to persistent and non-persistent data that is exchanged between tools that are running. Data exchange, then, is a measure of the amount of work that has to be done in order to exchange data between tools. A distinction between interoperability and data exchange is relevant because the two may use different mechanisms to implement the integration.

Synchronization is similar to data consistency. Where data consistency applies only to persistent data, synchronization applies to persistent and non-persistent data between tools that are running. Synchronization, then, is a measure of the extent to which cooperating tools communicate changes they make to shared, non-persistent data.

The goal of control integration is to allow the flexible combination of an environment's functions, according to project preferences and driven by the underlying software development processes the environment supports [22]. Control integration, then, addresses control transfer and service sharing issues. Control integration can be evaluated in terms of the number of services offered by a tool that are required by another tool in the environment. It can also be evaluated in terms of the number of services a tool uses that are provided by another tool in the environment.

The goal of Software Development Process integration is to ensure that tools interact effectively in support of a defined software development process [22]. Tools that make assumptions about the process in which they will be used may be difficult to integrate in sites where the assumptions are not valid (e.g. a design tool that assumes a specific design methodology).

The software development process is made up of a number of steps. Each step yields a result. Step integration is a measure of the extent to which tools work together to help achieve the desired result. An example of a collection of tools that are not integrated with respect to the software development process step follows.

A system processes a C++ source file into a C source file, compiles and links the C source, and then runs with a debugger that displays the C source file lines. This system is not doing as much as it might to help the user debug the C++ source file. The tools are therefore not integrated with respect to the software development process step.

Each step in the software development process is composed of a number of events that can or must occur. A precondition to using a tool may be the occurrence of some event (e.g. a unit test could be run after a successful compile and link). Event integration is a measure of the degree to which tools generate the events necessary for other tools to run.

Constraints may be placed on software development process steps or events. Constraints disallow progression toward steps or events that, if the constraint were not violated, would be allowed. Like data consistency integration and synchronization integration, constraint integration is concerned with tools working together within the guidelines of imposed constraints. Data consistency and synchronization are concerned with constraints on data. Constraint integration is concerned with constraints on functions provided by tools. In other words, use of a particular function in one tool may preclude use of a particular function in another tool. Constraint integration is a measure of the degree to which tools work together to provide appropriate functions in support of a software development process.

Brown and McDermid have defined tool integration in terms of the ability to record, share, and transfer information among tools [6]. Five degrees of tool integration have been defined. From low integration to high integration, they are: carrier, lexical, syntactic, semantic, and method.

The term "carrier" is borrowed from the field of electronic communications. Here it is intended to mean a common input / output format among all tools. Unix's file representation was the motivation for defining this level. All tools that are integrated at the carrier level have a common way to read and write data that they share.

A collection of tools that understand the format of the input / output are said to be integrated at the lexical level. These tools can use the same data or exchange data among themselves. A drawback for tools that are not integrated beyond this level is replication of code that parses or generates the input / output.

Tools that are integrated at the syntactic level agree on the rules used to format data that they exchange. These tools do not have to replicate code that parses or generates input / output, nor do they have to repeat actions to analyze, validate, and convert data.

At the semantic level, tools not only agree on the rules governing allowed data formats, they also agree on the meaning of the operations on the data. Tools can base their operations on the results of other tool operations without prior knowledge of the exact results.

Tools at the method level know the available data structures, the allowed operations on those data structures, and they know their role in the software development process. Method level integration provides the opportunity to:

- Constrain tool use to the correct point in the life cycle
- Offer guidance on what may be the best action at a particular time

- Automate any consequential actions (one tool can tell other tools about a change it made that will affect the data they both process)

Tools that exhibit team integration foster work performed by a group of people. This involves supporting both shared and personal data and tools, and providing methods for effective communication using the environment.

When tools exhibit management integration, management information is automatically derived from data and events in the system. This provides managers with accurate, up-to-date information leading to more informed decisions.

E. SOFTWARE ENGINEERING ENVIRONMENT SERVICES

The following paragraphs present services that a typical SEE would provide in order to accomplish the goals above. The goal that is accomplished is identified along with the description of the service.

The message services provided by the SEE framework allow tools to pass messages among one another in a manner that makes the location of the individual tools transparent. In addition, message services track the status of message delivery. These services provide support for control integration.

Digital Equipment Corporation, Silicon Graphics Incorporated, and SUN Microsystems Incorporated have formed a "CASE Messaging Alliance" in order to more rapidly achieve messaging standards. In October 1992, the alliance produced a document titled "CASE Interoperability Message Sets" that "contains high-level semantic – not syntactic – specification of messages in sets [24]." The document also defines requirements for the messaging services. These requirements bring insight into the types of services provided by this portion of the SEE Framework. Paragraphs that follow have been extracted from the aforementioned document.

The following capabilities must be provided within the messaging environment:

- The ability to deliver messages representing requests for functionality;
- The ability to deliver messages representing notifications of events;
- A mechanism or set of services for application message registration;
- A set of policies regarding delivery of messages;
- A mechanism or set of services to allow consistent delivery of a set of messages.

Message Delivery

Messages may be of two distinct types: Requests or Notifications.

Request messages are used to request that specific functional services be performed. An indication of the success or failure of the functional service must be provided to the requestor, as well as providing any additional reply required by the definition of the message.

Note – Functions that may be time consuming will return success or failure to indicate status of the initiation of a function. When the function actually completes the function will return a notification indicating the success or failure of the function.

Notification messages are used to provide notification to other applications in the CASE environment of events and state changes and do not require a response or affirmation of receipt.

The messaging environment must provide a reliable mechanism or set of services for delivering both these types of messages, as well as delivering the responses associated with the request messages.

Message Registration

The messaging environment must provide a mechanism or set of services to allow an application to define which messages it is capable of receiving. This registration capability should also provide the appropriate initialization services required for messaging to other applications in the messaging environment. Such services may cover the identification of the application to the environment, and the set up of communications between the environment and the application.

Policies for Message Delivery

The messaging environment needs to have a well defined set of policies that control the delivery of messages.

Consider the following algorithm for dispatching messages.

When a message is sent, the messaging environment will first:

1. Consider all currently available recipients of messages;

2. Finding no acceptable receiver, consider doing one of the following:

a. Auto-starting an application on behalf of the user

b. Send a failure message back to the requester, at this point the requester can make an intelligent decision as to whether to wait or report failure back to the user.

The auto-start capability enables an environment implementation to expand to fit the user's needs without user intervention.

Message Addressing

The messaging environment should provide a mechanism or set of services to enable one application to assure consistent delivery of a set of messages to a particular application or set of applications. For example, this would enable an application to have all of its edit requests serviced by a single editor instance, or enable all notifications to be sent to a select group of applications. Such a capability should be defined by the implementation, including the mechanism(s) to initiate, define the attributes of, and terminate the connection.

A possible implementation of this idea is to use a form of addressing that identifies a receiver by the function it performs. For example, an editor application would identify itself as "ascii text editor" so that any message that requests an ASCII text editor function could be handled by the application. This form of addressing also enables different applications to supply the same functions.

The user interface services provide sets of routines that perform all interactions with the user and form a foundation for a common user interface across all tools in the SEE. These routines will not, in themselves, force a common look and feel among all tools in the SEE. Application developers must follow style guidelines to achieve a common look and feel. User interface services support presentation integration.

Open Software Foundation Motif (OSF/Motif) represents a set of user interface services. As an explanation of the way OSF/Motif would be used by a developer, the OSF/Motif Programmer's Guide, includes Figure 4 [25].

Application developers would use services provided by OSF/Motif to create standard user interface objects. Developers also have the opportunity to use lower level

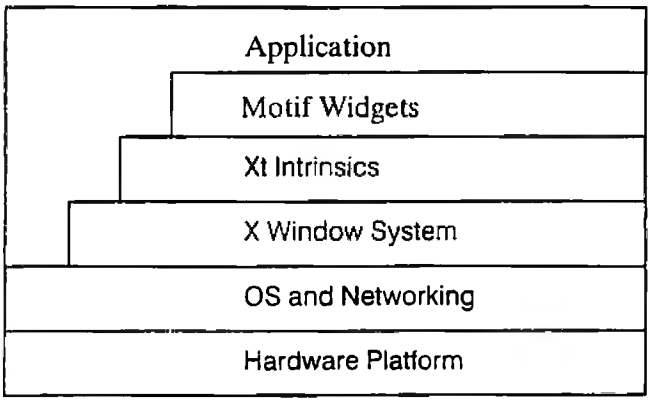

Figure 4. User Interface Development Model [25]

services that provide basic functions but do not attempt to provide common objects from the user point-of-view. As an example of the use of OSF/Motif, a typical application might produce a window that looks like the one in Figure 5.

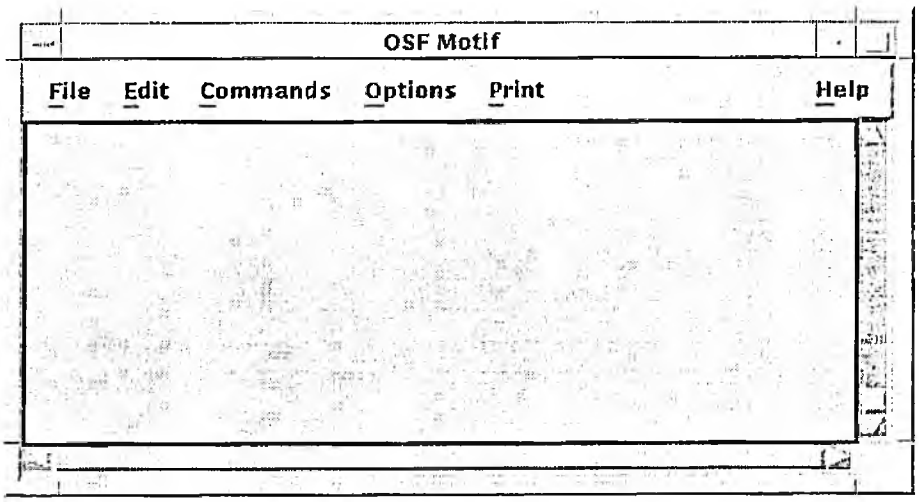

Figure 5. Sample OSF / Motif Window

Examples of the services typically provided by user interface services include:

- Ability to display predefined objects such as pointers, buttons, scroll bars, labels
- Ability to cause objects to respond to mouse and keyboard inputs or not
- Ability to group individual display objects
- Ability to display a named group of objects

The software development process management services manage the current state of the environment, and enforce software development process rules. The software

development process rules define valid states and the criteria required to transition from one state to the next.

Software development process management services support software development process integration. Typical services include the ability to define roles, the ability to define states and state transitions, and the ability to define the roles and events that are required to cause a state transition.

The object management services control access to all objects in the SEE. Each object may be a composite of many atomic elements. Examples of objects include: documents, source files, and problem reports. If all objects in the SEE are accessed via their object name, regardless of the method used to maintain them, then components of the SEE that access the objects are insulated from the specifics of database, library, or file access.

Object management services support data integration. Typical services include:

- Version control
- Check-in, check-out
- Copy, delete, rename
- Report generation

IV. COMMON GROUND

After covering the purpose, goals, and services of operating systems and environments in the last two sections, this section will narrow the focus to Distributed Operating Systems (DOS) and Software Engineering Environments. It will identify ways that the two might complement each other.

The motivation behind investigating distributed operating systems and Software Engineering Environments in the same context is performance. As we have seen in the last two sections, both systems are concerned with the transparent integration of system components. While SEEs are concerned with software components (application programs), distributed operating systems are concerned with both hardware and software components. However, both SEEs and DOSs are concerned with transparency from the system user and system programmer perspectives.

A. GOALS

Ultimately, distributed operating systems and Software Engineering Environments both seek to improve the productivity of their users. The two differ in the level at which they attack the problem, however, due to their role in the overall system. Their roles in the overall system are a natural result of their evolution.

Distributed operating systems have evolved from network and classical operating systems. Goals of the distributed operating system are oriented toward providing the necessary functions to allow efficient utilization of hardware resources for all current system users.

Software Engineering Environments, on the other hand, have evolved from application tool sets and stand-alone application tools. SEE goals are oriented toward providing efficient user services that enhance the productivity of a team of software developers.

The goal and service terminology in the previous sections was intentionally presented in a manner consistent to its presentation in other literature in order to emphasize the different perspectives to a similar problem space. While the goals presented in the previous sections continue to be important, the following paragraphs attempt to provide a single perspective for goals that are shared by distributed operating systems and Software Engineering Environments. The following goals address the user interface, transparency, executable process integration, team and management integration, performance, reliability, scalability, and efficiency.

The development environment should provide a consistent, intuitive interface. More specifically, equivalent operations should use the same commands and commands should be accepted that have meaning in the context of the user's software development process.

The development environment should appear as a single environment in which a team of developers and managers cause the project to progress through the software development process by performing specific operations. The user should be able to choose which development efforts are visible among any parallel development efforts. Specific software and hardware components used in an operation should not be visible unless visibility to this level is specifically requested by the user. A "user" may be a system user, or a system programmer.

The development environment should provide a "method level" of integration among executable processes. In other words, it should provide a level of integration such that the tools know the available data structures, and they know their role in the software development process.

The development environment should provide group communication mechanisms. It should provide levels of shared and personal data and executable processes. It should provide automatic collection of data relevant to the management of a software development effort.

The development environment should provide consistent performance for equivalent tasks. It should provide acceptable response and turn-around times.

The development environment should be available for use when required. It should recover from limited failure or, minimally, maintain data integrity.

The development environment should integrate new software development process steps, events, and constraints, new types of data, new tools, new hardware, or new versions of either with ease. It should allow any of the above to be removed or modified with ease.

With the exception of replicated data for improved performance, the environment should maintain one copy of every data item. It should not maintain a process in memory that can be reloaded at a later time without loss of performance.

B. SERVICES

There are some overlaps in the services provided by distributed operating systems and Software Engineering Environments. In table II, SEE service names are in the

left-most column and Distributed Operating System service names are in the middle column. The right-most column presents a single term that will be used to identify the SEE and DOS services in the remainder of this paper.

Overlap between SEE and DOS services occurs in the Object Management and Interprocess Communication services. With a slightly expanded object definition, Process Management and System services might overlap with Object Management services as well. While it seems likely that future operating systems will treat processes and system services as objects [26], that prospect will not be investigated here.

Table II. Service Terminology

SEE Service	Distributed Operating System Services	Combined Service Terminology
User Interface Services		User Interface Services
Software Development Process Management Services		Software Development Process Management Services
Object Management Services	Single, Global Protection Scheme Single, Global File System	Object Management Services
Message Services	Single, Global Interprocess Communication Mechanism	Interprocess Communication Services
	Single Method of Process Management	Process Management Services
	Single Set of System Calls	System Services

Distributed operating systems tend to provide support for files. They provide services to create, read, write, delete, and update files. They also provide services to protect files against unwanted access. All services, the files, and their attributes must be part of a global system.

Software Engineering Environments tend to be object oriented. While objects can be files or groups of files, they can also be database fields or groups of database fields. All objects must be part of a global object system, and must be protected by a global access mechanism.

Treating data as an object allows the underlying data structure to be handled as a logical entity. This eases the burden of maintaining tools that access the data, and tools can be used in many environments as long as the object format is the same, but there is a cost. An additional step is introduced with every data transaction. The extra step involves translating the generic object format into the actual data storage format.

For performance reasons, it would be best to place these services in the operating system. Due to the transient and site specific nature of the objects, however, it is not

reasonable to believe that all object services could be handled by the operating system. In the near term, it seems more reasonable to assume that object management services must be layered on top of operating system services. Object management services will also very likely rely upon a database. Tools that access objects in the Software Engineering Environment must not bypass the object management services. The object management services must have a well defined tool interface and must exhibit good performance.

Distributed operating systems must provide interprocess communication (IPC) across an entire network. The services are not limited to message passing. Global memory and remote procedure calls are two other possibilities. The DOS is responsible for locating the processes that are participating in the communication. The action of locating processes must be transparent to the participants. In addition, many to many communication must be supported.

The requirements of a SEE are similar to those of a DOS with regard to interprocess communication. However, because operating systems do not currently provide the services described above, current SEEs must provide the services they require themselves. Additionally, the object oriented nature of a SEE makes interobject communication the real issue for tools within the SEE. Interobject communication is another level above interprocess communication in that additional information is necessary in order to know whether a process known to the operating system represents an entire object or possibly just one method of an object. It is most reasonable to put the burden of interobject communication, should it be required, on the SEE and interprocess communication on the DOS.

If distributed operating systems can begin to provide efficient message services that make process location transparent, a large burden will be removed from current SEE implementations. In addition, expanding DOS IPC services so that they automatically start a process on the node that will yield the highest performance will lower maintenance of any SEE that attempts to perform that function now and will at the same time improve overall network performance. Some interesting points can be raised on this last issue. For example, it is likely that it would be beneficial for the DOS to know something about the way a process that it is loading will be used in order to make a good long term decision. Will the process be used interactively? Will it serve many other processes over a period of time or will it perform one operation and terminate? The SEE will be able to provide "educated guesses" to answer these questions. Thus, additional benefits can be realized if DOS designers look beyond some of the assumptions made in current operating systems and gain an understanding of the type of information that may be available from the SEE.

C. SUMMARY

Goals that are common to DOSs and SEEs emphasize that user interaction with the system should take place in a context with which the user is familiar; specific details of an operation including performance and reliability mechanisms should be transparent to the user; and the environment should be efficient, reliable, exhibit top performance, and accommodate the addition or removal of environment components.

Current SEEs must provide some services that have features not yet available in DOSs. When these features are added to the services provided by DOSs, a synergy will develop that will likely improve system performance, through non-redundant functional implementation at the proper level, and reliability, through dependence upon a single set of services with proven reliability. Improved user productivity will be the natural result.

V. SOFTWARE DEVELOPMENT PROCESS

Using the software development process as the basis for a SEE provides an effective mechanism to achieve the goals and services outlined in the preceding sections. Later sections will introduce new concepts that describe this in more detail. This section returns to the literature and focuses on the motivation for and definition of the software development process.

A. MOTIVATION

In every software organization, there is a defined way to generate the product. Granted, some organizations may be considered "ad-hoc," and probably have little to do with formally defined software development processes, but the fact that a term can be assigned to the way they do business means that it is defined in some sense. In an effort to allow organizations to assess their current process maturity and to provide a road map for continuous process improvement, the Software Engineering Institute (SEI) at Carnegie Mellon University created the Capability Maturity Model for Software (CMM) [3].

The CMM does not say how to define a software development process, but it provides an excellent tool for an organization to gain insight into its current processes. Using the CMM, an organization can identify which parts of their process are good, which parts are bad, and which parts are missing. The CMM defines five maturity levels for a software organization. A list of "key processes" is associated with all but the lowest level. The key processes associated with a particular level identify the issues that must be addressed in order for an organization to be considered to have that level of maturity. Levels build on one another. That is, in order to achieve level 2, you must meet the criteria for level 1 and the criteria for level 2. Figure 6 lists the key process areas for each maturity level in the CMM. Figure 6 also identifies both the name and number associated with each level, and the overriding characteristic required to move from one level to the next.

The SEI has also developed formal assessments and evaluations to support the CMM. After being assessed, an organization can potentially gain new customers by advertising that they are a level x organization. Intuitively, this makes sense. The maturity level acts as a sort of consumer's guide rating. Someone from outside an organization can get an idea of the processes followed in the organization by understanding the key processes associated with the levels identified in the CMM. They will feel confident that past success is an indication of future success. Use of the CMM has increased the visibility of the processes used to develop software.

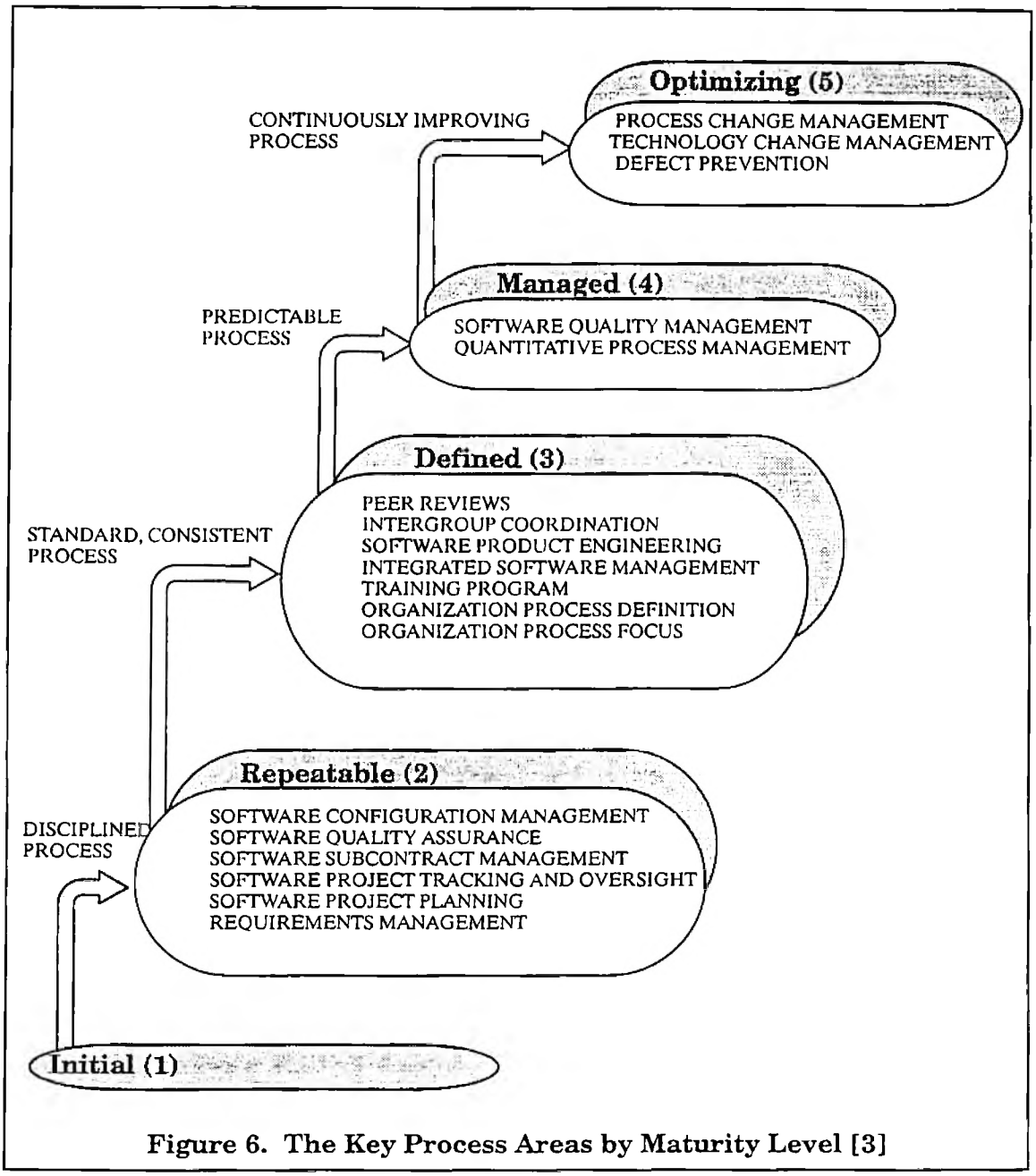

Figure 6. The Key Process Areas by Maturity Level [3]

The motivation behind this increased emphasis on software processes is stated in the first sentence of the CMM [3].

> After two decades of unfulfilled promises about productivity and quality gains from applying new software methodologies and technologies, industry and government organizations are realizing that their fundamental problem is the inability to manage the software process. [27]

In case the point is lost on potential contractors, the Department of Defense has started to require contractors to have a specific maturity level before they are allowed to submit proposals on certain programs. This challenges the vast majority of software

development organizations. Figure 7 shows the percentage of organizations in each level after the SEI had assessed 296 projects.

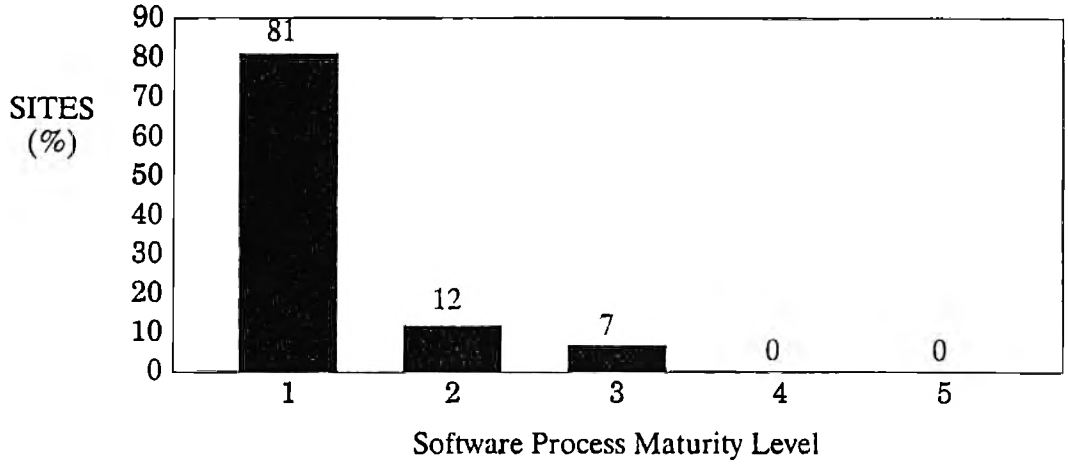

Figure 7. Percent of Organizations in Each Process Maturity Level [28]

As positive incentive, however, some organizations have willingly tracked the return on investment (ROI) when using process driven software development. These organizations did not use process driven software engineering environments, the ROI is strictly a result of improved process focus.

According to James Over [28], software quality improvements of 100 times are possible and not unreasonable on large projects; software productivity of 10 times are possible with larger gains projected; and a ROI of greater than 5 to 1 is possible. In other words, for every $15,000 to $20,000 invested, $100,000 is generated. The source of this information is the Juran Institute. Two specific cases support these claims [28]. Hughes Ground System Group documented savings of $2 million per year when using a process driven approach to software development. Raytheon documented that "every software initiative dollar invested in 1990 [to change to a process driven approach to software development] saved 7.7 project dollars."

In short, then, it seems that the benefits of process oriented software development are extensive. Since at least one major software customer, the Department of Defense, will not accept proposals without evidence of process maturity, it also seems that the penalty of not using a process oriented software development approach can be severe.

B. DEFINITION

It is worth noting, especially when considering the use of the software development process as a basis for providing automated assistance, that an important feature of a

process is that it provides some guidance and some tolerance when the process itself breaks down.

Processes are rarely foolproof, and adding more detail to a process so that every possible case is covered seems to increase the likelihood that a new case will be found that is not covered. There is a definite art to defining a process at the appropriate level. Generally, it should be accepted that there are portions of the process that can be defined to a level that they can be automated, if desired, and there are portions of the process that cannot. Such a simple statement seems obvious, but it would likely provide great benefit if it were reviewed daily by any team assigned to define or document the software development process for their organization.

The software development process, in the general sense, is "a set of activities, methods, practices, and transformations that people use to develop and maintain software and the associated products [3]." More specifically, the software development process can be characterized as a set of roles, responsibilities, tasks, task relationships, entry criteria, exit criteria, and products. The purpose of the software development process is to provide a basis for long-term productivity and quality improvement in a software organization. Instead of relying upon heroic personal efforts of a dedicated staff, the organization can repeat the same process they have used in the past, with minor improvements from lessons learned, and expect a successful development effort. The key is predictability.

Tasks are steps in the software development process. A task can be carried out by a computer or it can be carried out by a person. Task relationships define the order in which tasks can be started (enacted). Tasks may be enacted sequentially, or in parallel.

Tasks can be defined by states in a state diagram. Task relationships are defined by the state diagram. Each state represents a step in the software development process. Each transition from one state to another represents the completion of a step in the software development process and the start of a new one. The actual work is performed by a person, by a tool or by both while the SEE is in a particular state. When the work associated with a step in the software development process is complete, a transition is made to the next state where work related to that state can then begin.

It is worth noting that a software development process, when considered as a whole, will necessarily have work in progress in many states at the same time. The most obvious example of this might be a large project that has a number of developers. Some of the developers will be in the design phase while others may have completed design, completed implementation, and are now testing.

In addition, it is possible for a task to have subprocesses. That is, the process can be defined in a hierarchical manner. One task may break into one or more subprocesses. If a task is composed of a number of subprocesses, all of the subprocesses are considered to start concurrently when the parent task is started. This is a powerful mechanism for maintenance of the process itself, and for team and management integration. The process has many conceptual levels, any of which can be viewed at one time.

In order to clearly define responsibility and ensure quality work, access controls must be allowed on the states and on the transitions. The most natural way to control access seems to be through the use of "roles."

People or active SEE components that have been assigned to roles carry out tasks. Roles provide a convenient grouping and abstraction mechanism. Using roles reduces the coupling between the process definition and the individual people or active SEE components, allowing one to change without the other. A person or active SEE component may be assigned to multiple roles. Responsibilities are associated with a role. Assigning a person or active SEE component to a role assigns the corresponding responsibilities to them. Responsibilities include privileges to access SEE data or to execute SEE components. Responsibilities might also be defined as a requirement to perform a task. For example, the responsibilities of the Project Software Manager could be to review and approve documentation.

Some operating systems have what are called access control lists. An access control list is a list of the people that can access an object along with a specific indication of their access rights (e.g. (Jones:Read,Write,Delete; Smith:Read,Execute; others:none)). This mechanism is useful, but names of individual members of the organization do not belong in the software development process definition. The software development process is a definition of the work that is performed, the dependencies of the work, and the roles of the people authorized to perform the work and verify its quality. With this in mind, it is natural to replace an individual name, such as Jones, with a role, such as Developer when defining access control related to a software development process.

Assigning a role to a state defines the type of people that can perform the work associated with that process step. Likewise, assigning a role to a state transition defines the people who are authorized to declare that work has been satisfactorily completed in the current state and work in the next state may now begin. Prior to performing any work in a process driven SEE, a person must be assigned to their appropriate role(s). In other words, prior to performing a requested action, the SEE will verify that a person has been authorized to perform the action by comparing the role assigned to that person in this part of the software development process with the list of roles authorized to perform the action.

A simple example of a development process is shown in Figure 8. This example shows a portion of the development process that might be used for source code. Typically, source code evolves from being "in work" to being "under test." After testing, it is either declared "tested", if it passes the tests, or if it does not pass the tests, more work has to be done so it is again "in work." In Figure 8, the states In-Work, Under-Test, and Tested, along with the relationships indicated by the arrows, are meant to reflect this activity. In-Work is the start state and Tested is the final state.

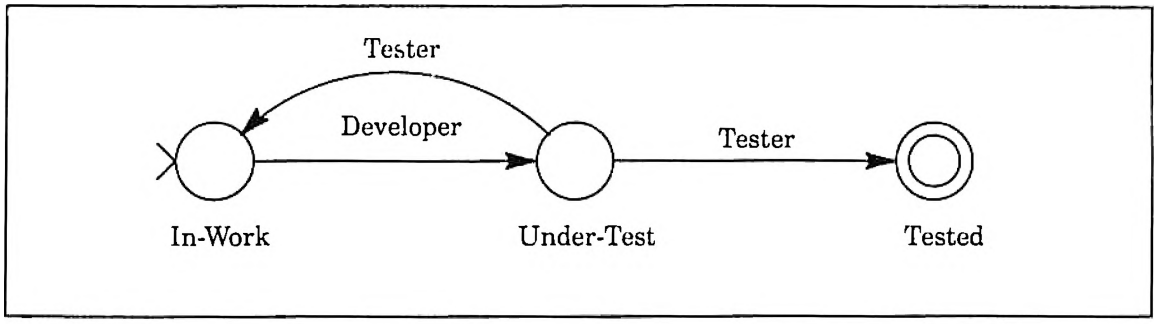

Figure 8. Sample Development Process

Figure 8 also shows roles assigned to each state and transition. Even with this simple example, it can be seen that while a Tester can place source code into the In-Work state, only a Tester that is also a Developer can work on it. By ensuring that no Tester is also a Developer, a degree of control is established.

This method works at a high level, but could turn out to be somewhat awkward in practice. A better implementation would allow a role to be used as the default access control list, but also as the set from which individuals may be selected to establish a more restrictive access control list. In other words, a role would define the set of people that may perform certain work, however, for particular objects, the list may be further restricted.

An implementation that would support this mechanism would allow a specific individual to be assigned as the Developer for a specific source file. While the software development process simply defines a Developer as someone that can work on source code, the SEE allows the list of developers to vary for each source code file. Jones, then could be assigned as the Developer of a particular source code file. Smith could be assigned as the Developer of a different source code file, but could also be assigned as the Tester for Jones' source without any fear that Smith would find a problem in Jones' source and take it upon himself to correct Jones' source. Smith was not assigned as the Developer of Jones' files and therefore can not access them in the In-Work state.

Each task has an associated set of entry criteria, exit criteria, and products. The set may be empty. Required products may be incorporated into the entry or exit criteria. A task cannot be enacted until the preceding tasks have completed and the entry criteria are satisfied. A task can not complete until the exit criteria are satisfied. Products will be produced during task enactment and may be available before the task completes.

VI. PROCESS DRIVEN ENVIRONMENTS

While other process driven environments may exist or be in development, there are currently two process driven efforts that are notable for their maturity. One is a SEE developed by the University of Southern California (USC) [4], the other is a "process engine" developed by Hewlett Packard (HP) [29]. Features of both are described below.

Both approaches use a process engine. The process engine is responsible for following the defined software development process, spawning tasks when appropriate, and keeping track of the status of tasks. HP calls their product (SynerVision) a process engine, but their definition is a little broader and includes things like a user interface. The process engine is really the primary component in a process driven SEE. It is responsible for all progress through the process.

The USC approach has two distinct and separate user interfaces, a developer interface and a manager interface. In the USC system, a developer enacts the tasks in the software development process while the manager controls them. Each user interface supports the specific responsibilities of the roles expected to use it. The USC approach uses multiple windows.

The HP approach has a single user interface for all roles. While developers and managers perform different functions, roles are used to restrict the actions. The HP interface basically presents the software development process as a textual hierarchy of tasks (each task on a new line with each level in the hierarchy indicated by indenting the line from the line above). The HP approach uses one window.

The USC approach defines their software process as "a collection of objects representing activities, artifacts, tools, and developers [4]." Like this paper, the term "task" is used to represent a step in the software development process. Tasks are broken down into other tasks and actions. Actions have four attributes assigned to them that can be used to define products and roles. These attributes are: agent (equivalent to "role" in this paper), required resource, tool, and provided resource.

The HP approach defines the software development process in terms of a hierarchy of tasks and subtasks. Roles and products are assigned to tasks or subtasks. Both people and tools can be assigned to the roles. Other attributes of tasks are user definable. Given the appropriate conditions, tasks that have tools assigned to their associated roles will execute automatically.

VII. BENEFITS OF THE SEE

Given the background on software development processes and on current process driven SEEs presented in previous sections, this section will now explore the benefits of a process driven SEE. Aspects of integration and performance are considered.

A. THE SOFTWARE DEVELOPMENT PROCESS AS A FOUNDATION FOR THE SEE

The portions of the software development process that can be automated are not the only useful portions when considering the use of software development processes in a SEE. In fact, the primary motivation for a process driven SEE is the provision of context.

In an organization with a well written software development process, any developer could point to a task in the process to identify the work they are doing. They could point to a role to identify their responsibilities relative to that task. With that information alone, they have revealed a lot of information about what they are currently doing. A look at the entry criteria for the task reveals important information about work already performed. A look at tasks that preceded this one reveals more. A look forward provides an indication of the tasks remaining. A look at the other roles assigned to the current task reveals whether the person is working alone or as part of a team. A look at the membership of all the roles for the entire process provides an indication of the size of the team. It should be obvious that this list could continue to some length. The point is that the process provides the context within which the tasks and roles have meaning. Given the process and an indication of the tasks that are current, a lot of information is revealed about where the development effort is and where it is going. Building the ability to use this information into a Software Engineering Environment means a large difference in the amount of assistance the environment can provide.

Instead of an environment that blindly does the work it is instructed to perform, the environment has the "playbook" and knows what to expect and what is allowed. A high degree of process integration in a SEE provides a strong foundation for team integration, management integration, and control integration. It also provides a strong method of improving system performance.

B. TEAM INTEGRATION

All individuals on a development team are assigned a role. Every task and task transition has a role assigned to perform the work in the task or to authorize the

transition to the next task. Storing this information in the SEE provides a powerful reference for manual lookups or automated actions. The environment, for example, can mail a message to the individuals assigned roles in the next task when a current task is complete. Teammates can easily communicate with other members of specific teams without the need to maintain a number of potentially out of date mail distribution lists. The environment is known to have one up to date list of the members of each team as defined by the software development process. Communication could be performed using role or team names in order to enhance the ability to reach just the right people with minimal effort.

Communication among team members is also fostered through the fact that everyone has visibility into the current state of the development effort. Everyone has the ability to find out exactly what everyone else is doing.

C. MANAGEMENT INTEGRATION

Proper data for accumulation in management reports can only be extracted with knowledge of context from the perspective of the software development effort as a whole. The software development process provides this all encompassing view of the software development effort. Tools can collect specific metric data when they operate within the context of a defined process.

Because the entire development process is recorded in the environment, and the environment knows which tasks are currently in work, a manager can quickly get a feeling for the status of a project. Resources may be juggled if one person characteristically has a large number of items in work while another typically has few. Schedule progress can be tracked automatically by defining the schedule in terms of the tasks of the software development process. The environment would simply keep track of the tasks that are currently in work so the actual data on the schedule is just a "pretty print" of this information.

D. CONTROL INTEGRATION

Interface standards are an important issue when considering control integration. Tool interfaces must be designed in such a way that tools can work cooperatively. Mapping the tools onto a software development process, identifying which tools are used to perform which tasks, provides some insight into the types of information required in an interface standard. For example, a process might identify that a lines of code metric is to be collected every time a source unit is compiled. If the compiler is the tool that also generates this metric, but it does so optionally, then it is important for the interface standard to allow this option to be specified.

One method for doing this, and one which provides strong tool to tool control integration as well is to use the process as a foundation for control integration in the same manner that Diana has been used as a foundation for data integration in environments such as the Rational Environment™. Diana is an "intermediate" source code representation that maintains syntactic meaning [30]. Tools that reference this representation are immediately provided with not only the basic data, but attributes of the data as well. So, for example, a tool does not have to parse an entire file, or collection of files, in order to find the place where two particular variables have been used in the conditional portion of an if statement. The Diana representation retains information about the components of the source units so the tool does not have to regenerate that information by parsing the files. The tool still has to look through the Diana representation for the specific combination of attributes desired to meet the criteria, but this is a much more manageable task.

Using the software development process as a foundation for control integration would mean the process would define the task that specific tools would perform, and it would define the way that tools would work together in order to accomplish their tasks. As a small example, a SEE might allow a software developer to drag and drop a source file from the configuration management system to an icon representing the compiler. The context provided by the software development process might tell the configuration management system that the file should be fetched, but not reserved. It would tell the compiler that it should generate metric data. It might also automatically cause a metrics collection tool to gather the data generated by the compiler and store it with official metrics for the project on which the developer is working.

While these steps could be automatically performed in some manner without the use of an underlying process, the process provides the additional capability for the environment to modify its behavior based upon the current position within the process. It should be additionally noted that the user must have the ability, possibly within constraints, to override the default actions defined for each task.

E. PERFORMANCE

As noted previously, one of the major issues that seems to be unresolved with regard to performance, is that of having to load, or in some way prepare, a tool for use. Typically, system administrators will define the collection of software that is "resident" in memory when a computer boots. Some guidelines are used when defining this list, but two of the major factors are the amount of use the product is expected to have and the amount of time it takes to load into memory. This has proven an effective method of addressing the performance slow down when a new tool is requested, but there are some important new considerations to make that reduce the effectiveness of this method.

The cost of some tools used in SEEs has driven most companies that use the tools to purchase network licenses. A network license is one which allows the tool to be used concurrently by a limited number of users regardless of their location on the network. In other words, the software is available for use anywhere on the network but is not loaded onto a machine until it has been requested there.

While the system administrator can not help in this scenario, strong process integration in a SEE can. With strong process integration, the SEE knows what tasks are current and what tools are required to support them. Additionally, it knows what tasks are likely in the future. In addition to mailing a notification to individuals assigned to a new task when a current task is complete, the SEE can load the tools necessary for the new task if they are not currently available. It would be beneficial in some cases to load tools for the next task prior to completion of the current task. The SEE could use some criteria such as "when 80% of the subtasks have completed their assignment, load the tools for the next task."

F. SOME GUIDELINES

Even a high level knowledge of context can help. For example, if the testing tool a software developer is using has been initiated by the environment to know that it is operating in the design phase and not the formal test phase, for example, the tool would not automatically record test results in the formal log. When the tool is used during the formal test phase, it would automatically record the results in the formal log.

This example illustrates that use of the software development process is not something that automatically means restrictions are imposed. To the contrary, the emphasis should be placed on using the process to prevent mistakes and provide automated assistance when possible. Restrictions should not be imposed without justification. Reducing the freedom of the software developer can reduce their familiarity with the software and thus actually reduce quality.

That being said, restrictions do have their place. It would not be wise to allow an overambitious tester access to a developer's source code. It would also not be wise to allow the developer who is testing during the design phase to record their test results in the formal log, even if they so desired. Their records can be kept in their own workspace, and a SEE that has knowledge of the software development process can automatically support this. Recording unofficial results with official ones may be a short term convenience for a developer, but will likely distort project metrics and lead to incorrect conclusions. It is important to give developers the flexibility to do their job, but it is just as important to protect the development effort by preventing them from violating the rules outlined in the software development process.

G. INTEGRATION WITH THE ORGANIZATION

As discussed above, it is important for a SEE to provide an interface that allows users to perform work in a context with which they are familiar. This type of interface is more intuitive, allowing implementation details to be more transparent. A high degree of process integration can support this intuitive interface and lead to a high degree of control integration. It also supports control, team, and management integration.

In an organization where the software development process is highly integrated with the SEE, the foundation is in place to view the software organization as a part of the larger organization. That is, in a software organization where individuals can be viewed within the context of their roles on a development team, and development teams can be viewed within the context of their roles within the software organization, the basic elements exist to allow the software organization to view itself within the context of the larger organization. Chen and Norman use Figure 9 to show the levels of support provided by a SEE in relation to the activities of an organization [8]. Figure 9 provides a starting point, then, for defining the roles of the software organization within the larger context.

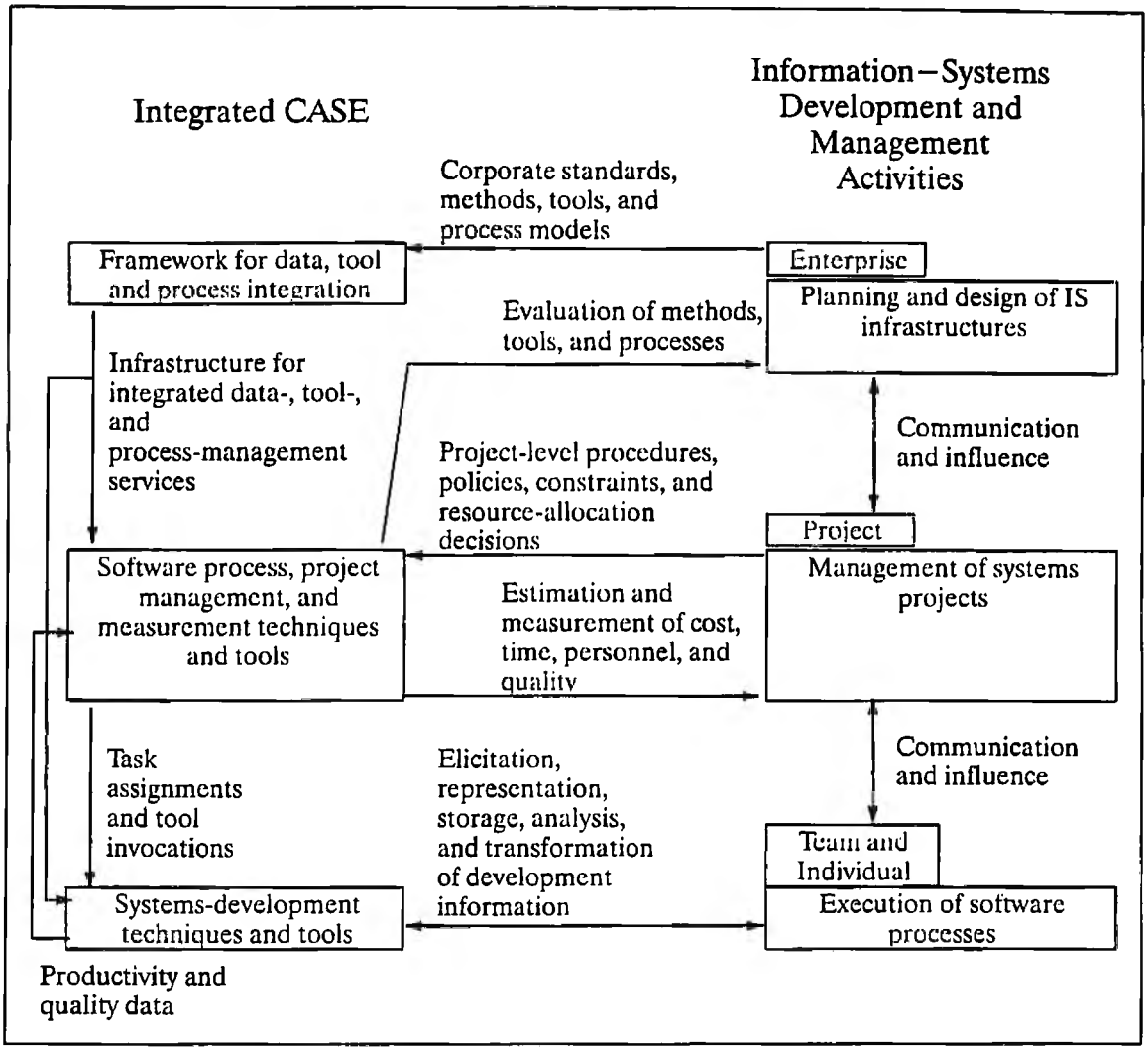

Figure 9. Placing CASE Tools in the Context of an Organization

VIII. PROCESS COMPONENTS OF THE SEE

This section introduces components of a SEE that provide software development process integration. A SEE with these components would support the goals of control, team, management, and process integration introduced in section III and would provide the benefits outlined in section VII. The SEE would be scalable and provide high performance.

A. AUTOMATING THE SOFTWARE DEVELOPMENT PROCESS

The software development process must be defined in the system in a manner that will allow components of the system to act based upon that definition. The recommendation here, in brief, is to assign an active part of the SEE, an executable unit referred to as an "agent," to manage each task in the software development process. When an agent is created by the operating system, the appropriate data structure is placed in its portion of memory that defines the portion of the software development process relevant to the task assigned to that agent.

More specifically, the current task definition, parent task definition, all child task definitions, and the address of a binder would be provided to the new agent upon creation. The information supplied to the agent would come out of a master context repository and would be provided by the operating system. This mechanism is similar to the Unix "fork" operation or the VAX VMS "spawn" operation. The difference is that instead of the operating system creating a new executable process with a context that is like their parents', the operating system would create a new executable process with a context based upon the position of the logically associated task within the software development process.

Upon creation, the agent would define its own "mailbox" or "pipe" for communication with other agents. It would then register the address of its mailbox with its binder to indicate that it was ready to begin work.

The following paragraphs provide more specific information about each major SEE component. Figure 10 provides a pictorial view of the major components defined below. Arrows indicate lines of communication between the components.

1. Agents. Basic to the implementation proposed here is the concept of an agent. An agent is an executable unit that is an active component in the SEE. Although the previous paragraphs described an agent as an executable unit assigned to manage a task

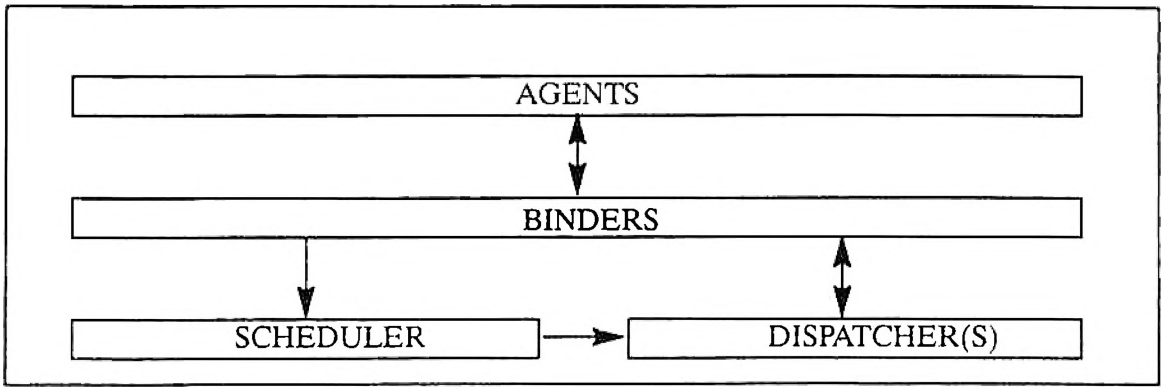

Figure 10. Major Components Supporting Process Integration

in the software development process, there are really two types of agents: a task agent and an application agent.

The task agent is the type that was previously described. It is an active part of the SEE that directly supports the execution of a task in the software development process. The existence of a task agent is evidence that the associated task in the software development process is either in work or is expected to be in work soon. Remember that in order to improve performance, tools may be pre-loaded or retained if they are expected to be used soon.

An application agent is responsible for a particular executable that has been or is likely to be requested by a task agent in order to accomplish its task. The executable managed by the application agent might be a robust tool or some small utility. The application agent oversees loading and unloading of the executable and translates software development process information into commands that the executable understands.

Basically, when a task in the software development process is started, a task agent is created to manage it. Task agents are generic in function and operate based upon the attributes of their associated task. When a tool or other executable in the SEE is started, an application agent is started to manage it. Application agents are not generic but are tailored to support the particular executable that they manage.

Agents communicate with other agents in a manner that makes their actual location on the network immaterial. Agents communicate in order to accomplish software development process tasks in a cooperative manner.

Like objects (in object oriented technologies), all agents are associated with a class. The class defines the way the agent will respond to requests from the environment.

Responses are defined in terms of "methods" that are executed when the agent is presented with specific "events." Methods assigned by virtue of inclusion in a particular class can be overridden by associating a different method with any software development process task. When the associated task agent is created, it will be assigned the method(s) associated with the process task in place of the method(s) provided by the class. (The manner in which agents operate was inspired by both encapsulation mechanisms such as those present in HP SoftBench, and by Frame-based Artificial Intelligence Systems [31].)

When a task agent is started, it references its associated task to determine what to do. If it finds that its associated process task is comprised of subprocesses, then the task agent starts those subprocesses by starting other task agents to manage them. The first task agent is considered the parent of the other task agents. The parent task then communicates with its subtasks as required to respond to requests from the environment.

It is also possible for a task agent to find, upon starting, that it will require some number of executables in order to accomplish its task. The task agent would then locate and register with the appropriate application agents that are managing the executables that it requires. Registration is effected so that the application agent knows what agents are using its services. An application agent that does not have any other agents registered will turn itself in for termination.

2. Binders. Agents do not automatically know the addresses of all agents with which they will communicate. In order to locate another agent, the first agent would make use of a SEE component called a binder. A binder is another active component in the SEE. It is responsible for forwarding agent requests to agents that it believes can best handle the request. For performance reasons, there are a number of binders in the SEE.

Binders maintain a registry of agents and other system components. Every binder registers every other binder, but each binder only handles specific classes of agents. A binder that registers agents of a particular class will register all agents of that class that are currently active in the SEE. If a binder receives a request targeted to an agent that it does not register, it will forward the request to a binder that does register that class of agent. Assuming an agent exists to receive the message, the message suffers a delay of at most two binders along the way to its final destination.

3. Scheduler. In order to attempt to ensure that an application agent and its associated executable already exist at the time they are requested, some task agents will request that the application agents are started prior to their actual need. With a number of task

agents potentially making these requests at the same time, a focal point for coordination is required. Of course, agents and executables can not limitlessly be loaded into memory either, so a decision must be made to remove agents and executables. The scheduler is responsible for deciding which new tasks to make available and which existing tasks can be removed from the system. Note that the tasks are not actually loaded or removed by the scheduler. In the case of task creation, the scheduler sends a request to the dispatcher. In the case of task removal, the scheduler assigns priority rankings to those agents that are not currently in use. The priority defines the likelihood that an agent and executable, if applicable, will be requested in the near future. In general, the lower the likelihood, the higher the chance for removal. The scheduler can determine the likelihood of being requested through knowledge of the current task, upcoming tasks, and the executables required to support them.

4. Dispatcher. The dispatcher actually loads new agents and removes existing ones from the system. The dispatcher attempts to allocate SEE components across the network in a manner that will provide peak performance. When the scheduler decides it is time to load a new agent, the dispatcher decides where to put the agent and its associated executable, if appropriate, on the system. The dispatcher itself decides when an application agent and executable should be removed and which one will be removed. It does this based upon the priorities assigned by the scheduler and rules designed to balance the network load.

The dispatcher is well suited to be a component of a distributed operating system. It would fit well as a "Process Management Service", in the operating system sense, defined in section IV.

B. PERFORMANCE

Current process driven SEEs have a process engine that references a repository to manage the tasks associated with a software development process. The SEE components introduced above distribute the work of the process engine across the task agents in the SEE. Task agents are created with knowledge of their position in the software development process. Once an agent establishes contact with another agent, it registers the address of the other agent so that future contact will be direct. Figure 11 shows a diagram of direct agent to agent communication. The figure shows that each agent has a mailbox to queue messages in case it can not process messages as fast as it receives them.

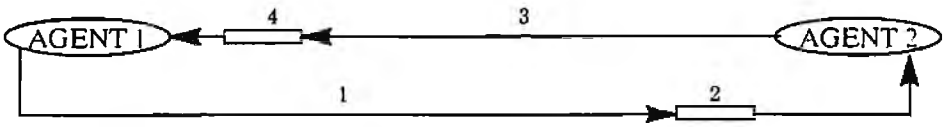

Figure 11. Best Case Agent to Agent Communication

Without considering the performance of individual tools in the SEE, a number of different performance analysis could be performed. Questions such as the following could be asked:

- How many binders should there be relative to other agents?
- How should binders be distributed across the network? One per machine? One per cluster?
- Should binders register agents based upon class or location on the network?
- How many binders should register a particular class of agent?
- What policy should be used by a binder when it selects another binder to receive a request that it can not process? (If multiple binders register agents of one class, it is likely that one of the binders that registers that class will process the request faster than the others.)

Answers to these questions are probably best obtained through simulation. Mathematical formulae could be developed to account for average queue behavior over time, but the average case is by definition a compromise. The scheduler component of this SEE has the ability to dynamically adjust the configuration of the environment by recognizing the current demands and likely future demands through knowledge of its position in the software development process. Accurate simulations could determine the rules that the scheduler component would use.

1. Scenarios. Given that good rules have been developed for the scheduler, the worst case performance behavior will come in the scenarios that follow. The scenarios are presented according to increasing deviation from the best case of direct agent to agent communication. Each scenario is accompanied by a figure that shows the SEE components and the performance factors that must be considered. SEE components are identified by named ovals. Each SEE component has an input queue (mailbox) from which it receives messages. The input queue is shown as a small rectangle in each figure. Message passing between agents is shown as a solid arrow. An agent starting another

agent and its associated executable is shown as a dashed arrow. Performance factors are represented by numbers next to the input queues, solid arrows, and dashed arrows. Performance factors under consideration are the time a message spends waiting to be processed, the propagation delay required to transmit a message from one agent to another, and the time required to prepare an agent and its associated executable to be available for execution. Performance factors are numbered in sequence.

In each figure, the SEE components and performance factors that must be considered over and above the best case scenario are identified by placing them on a grey background. On each scenario after the first, performance factors that cause overall performance to degrade from the previous scenario are circled.

Table III provides an overview of the scenarios. It shows the conditions of the system that affect performance and identifies the scenario that discusses the corresponding SEE actions. An "x" in table III represents a "don't care" condition. That is, due to other conditions, the condition in that column will not affect the actions taken by the SEE components. Table III is arranged for readability. The scenarios are numbered from best performing (scenario 1) to worst performing (scenario 6).

Table III shows that it is possible for the address of the agent to which a message is being sent (the target agent) to be known, and yet the agent is not available. This is an error condition. It could be the result of a machine crashing, a break in the network cable, or other reasons. Regardless of the reason, in this error condition, the message will be returned to the agent with an error flag set. The sending agent can then decide whether to send the message to another agent, or it may respond in a different way.

It should be noted that this condition is theoretically possible any time a message is sent from one SEE component to another. A message from one binder to another may not reach its destination because the second binder is inaccessible due to some atypical condition. The following scenarios only consider the normal case of successful communication.

Table III. Scenario Overview

Is Address of Target Agent Known?	Does Binder Register Target Agent?	Is a Target Agent Available?	Is a Target Agent Being Initialized?	
Yes	x	Yes	x	Best Case
Yes	x	No	x	Error (see text)
No	Yes	Yes	x	Scenario 1
No	Yes	No	Yes	Scenario 3
No	Yes	No	No	Scenario 5
No	No	Yes	x	Scenario 2
No	No	No	Yes	Scenario 4
No	No	No	No	Scenario 6

Scenario 1

Conditions: The first agent requires services from an application managed by a second agent. The second agent and application are available but not yet known to the first agent. The binder of the first agent has already registered the second agent.

Environment actions: The first agent must send its request through its binder since it does not know the address of an agent that can service its request. Note that the request is typically an actual service request by the first agent, it is not simply a request for an agent of a particular class. The binder finds an available agent (the second agent) in its registry and it forwards the request directly to the second agent. The second agent processes the request and replies to the first agent.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is the time spent in one additional queue (2) plus one additional propagation delay (3). Figure 12 shows all steps in the scenario.

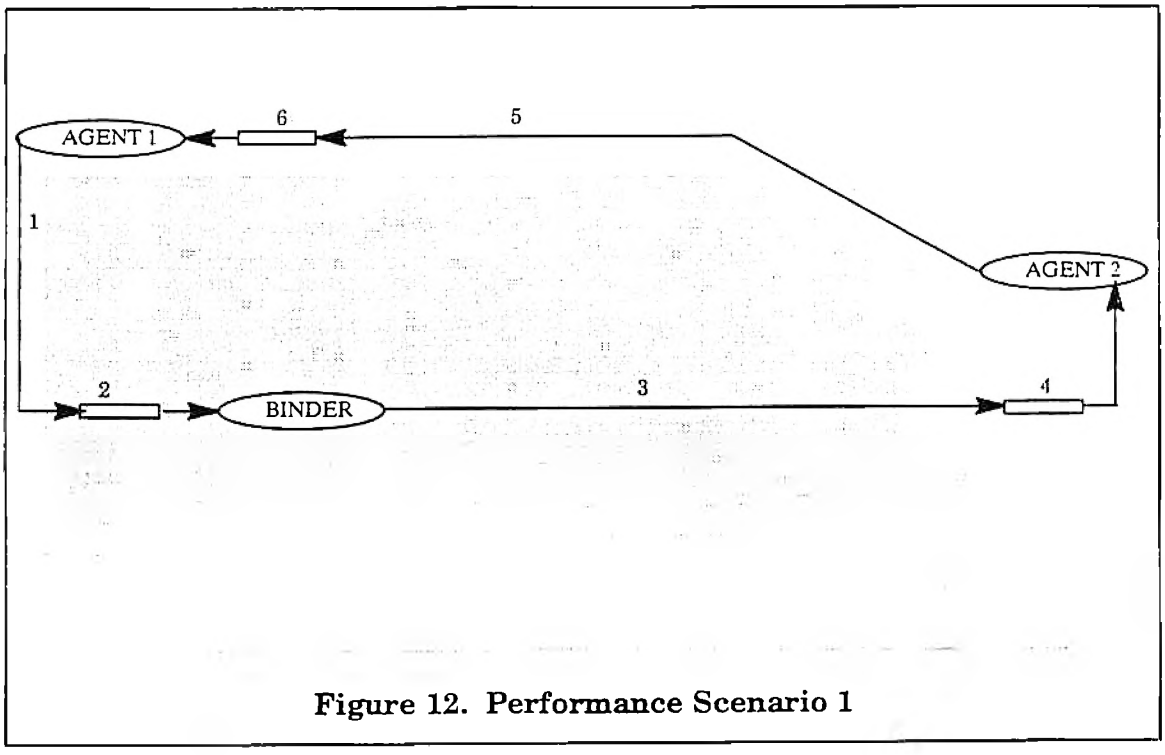

Figure 12. Performance Scenario 1

Scenario 2

Conditions: The first agent requires services from an application managed by a second agent. The second agent and application are available but not yet known to the first agent. The binder of the first agent does not register agents of the desired class.

Environment actions: The first agent must send its request through its binder since it does not know the address of an agent that can service its request. The binder of the first agent does not register this class of agent, so it must look up the address of a binder that does and forward the request to that binder. The second binder will receive the request, find an available agent (the second agent), and forward the message directly to the second agent. The second agent processes the request and replies to the first agent.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is the time spent in two additional queues (2,4) plus two additional propagation delays (3,5). Figure 13 shows all steps in the scenario.

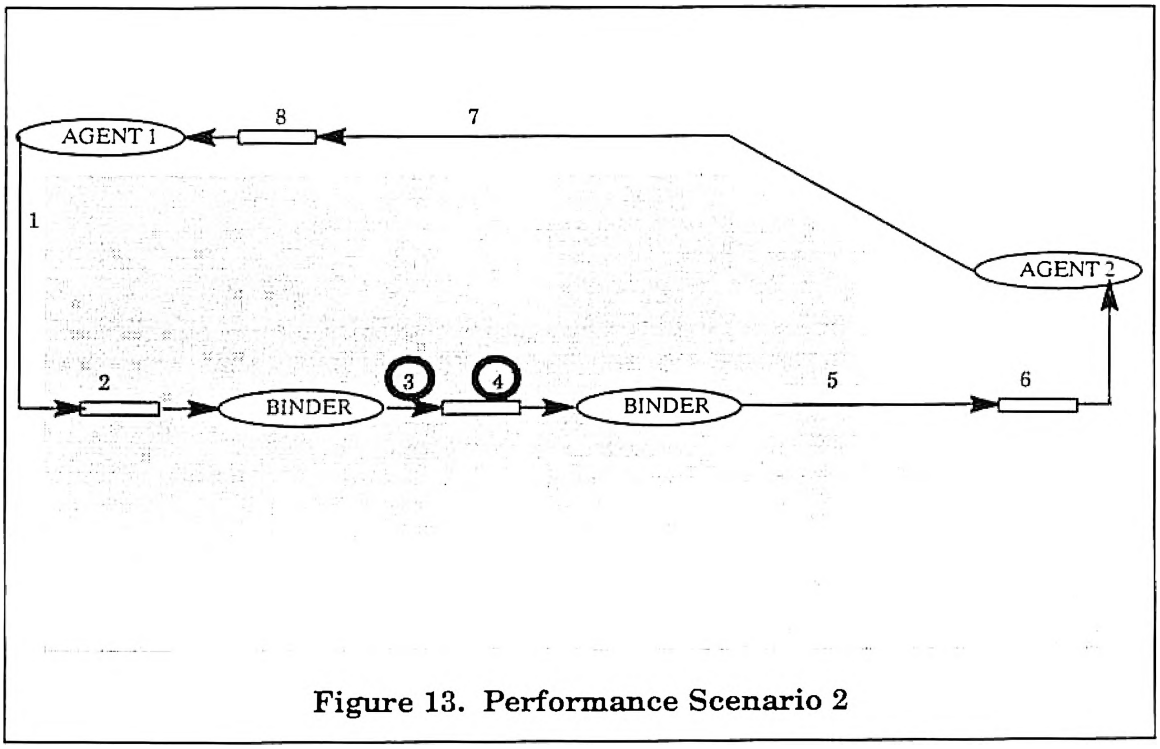

Figure 13. Performance Scenario 2

Scenario 3

Conditions: The first agent requires services from an application managed by a second agent. The binder of the first agent registers the agents of the desired class but the second agent is not available. The second agent is currently initializing.

Environment actions: The first agent must send its request through its binder since it does not know the address of an agent that can service its request. The binder of the first agent recognizes that it registers agents of this class, but it does not have any agents registered. It must store the request and forward a request to the scheduler to load a new agent. The scheduler determines that an agent of this class is already initializing, so it does not forward the request. The dispatcher has already started the new agent. Once the new agent has initialized, it sends a message to its binder to register. The binder of the new agent then sends a message to all binders that register agents of this class. The binder of the first agent is one of these. When it receives the registration, it looks for any messages awaiting an agent of that class, finds the message sent by the first agent, and forwards it to the new agent. The second agent processes the request and replies to the first agent.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is 4 additional propagation delays (3,6,8,10), plus the time spent completing the load of the new agent (assume 0.5 of the total load time)(7), plus the time spent in four queues (2,4,7,9). Figure 14 shows all steps in the scenario.

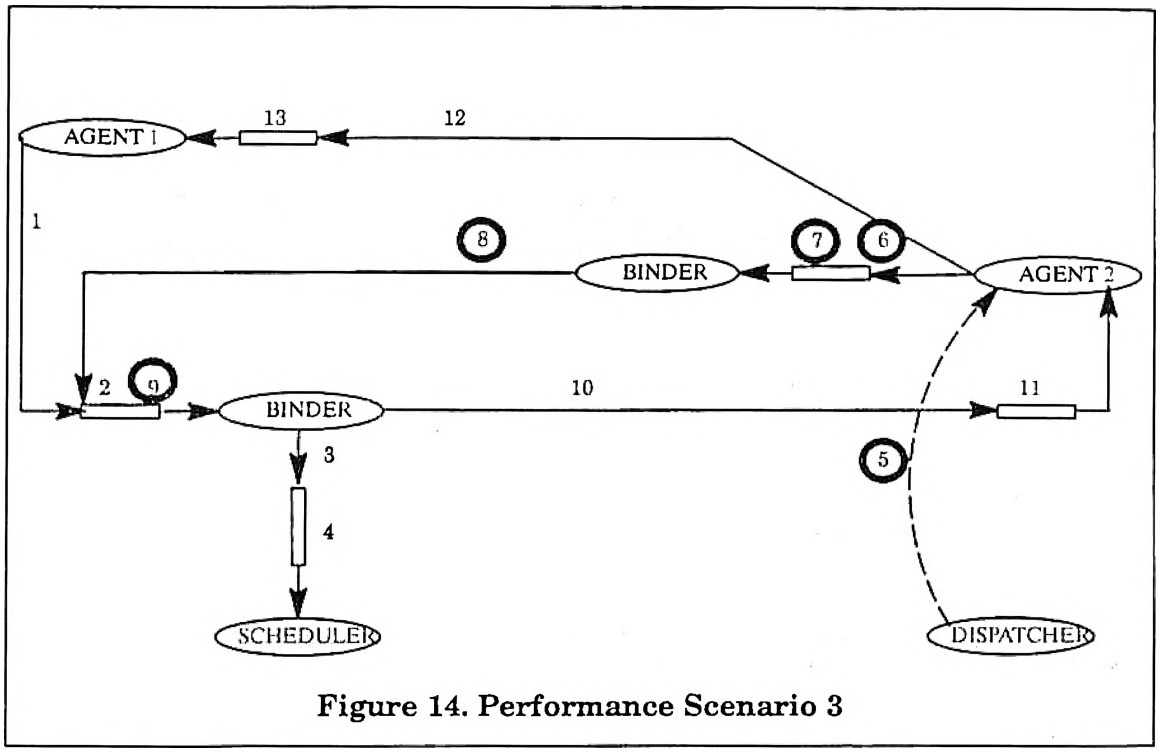

Figure 14. Performance Scenario 3

Scenario 4

Conditions: Same as scenario 3 except that the binder of the first agent does not register agents of the desired class.

Environment actions: The binder of the first agent does not register this class of agent, so it must look up the address of a binder that does and forward the request to that binder. The second binder will receive the request, recognize that it registers agents of this class, but it does not have any agents registered. It must store the request and forward a request to the scheduler to load a new agent. All other actions are described in scenario 3.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is five additional propagation delays (3,5,8,10,12), plus the time spent completing the load of the new agent (assume 0.5 of the total load time)(7), plus the time spent in five additional queues (2,4,6,9,11). Figure 15 shows all steps in the scenario.

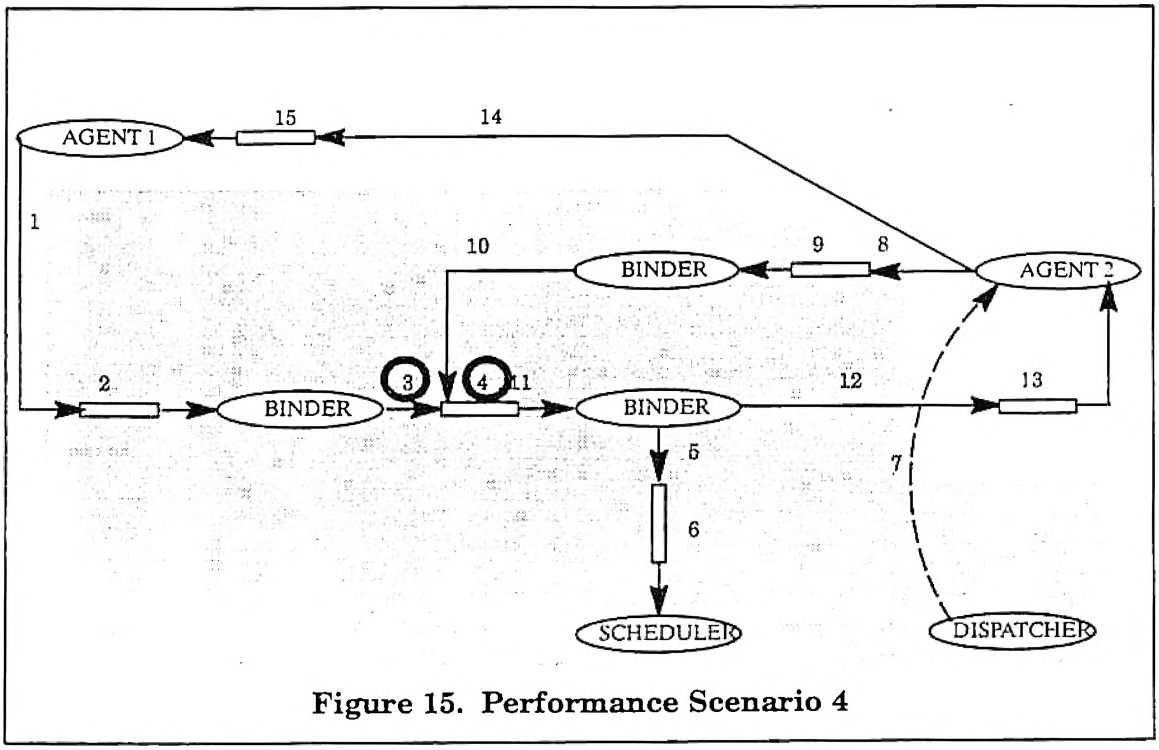

Figure 15. Performance Scenario 4

Scenario 5

Conditions: Same as scenario 3 except that the scheduler determines that it must request a new agent to be loaded.

Environment actions: The scheduler must send a message to the dispatcher. The dispatcher receives the message, determines the best location for the new agent, and initiates the load. All other actions are described in scenario 3.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is five additional propagation delays (3,5,8,10,12), plus the time spent completing the full load of the new agent (7), plus the time spent in five queues (2,4,6,9,11). Figure 16 shows all steps in the scenario.

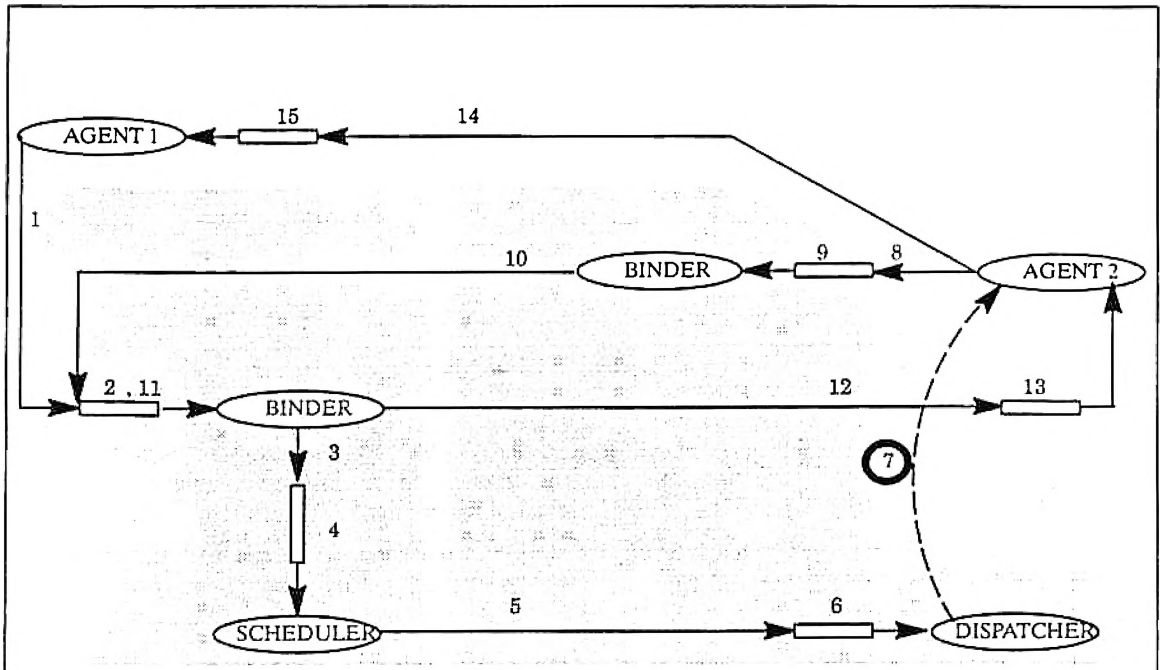

Figure 16. Performance Scenario 5

Scenario 6

Conditions: Same as scenario 5 except that the binder of the first agent does not register agents of the requested class.

Environment actions: The first binder must determine that it does not register agents of the requested class, find the address of a binder that does, and forward the message to that binder. All other actions are described in scenario 5.

Performance vs. best case: Assuming all propagation delays are approximately equal and processing time within a component is negligible, the total delay versus the best case is six additional propagation delays (3,5,7,10,12,14), plus the time spent completing the full load of the new agent (9), plus the time spent in six additional queues (2,4,6,8,11,13). Figure 17 shows all steps in the scenario.

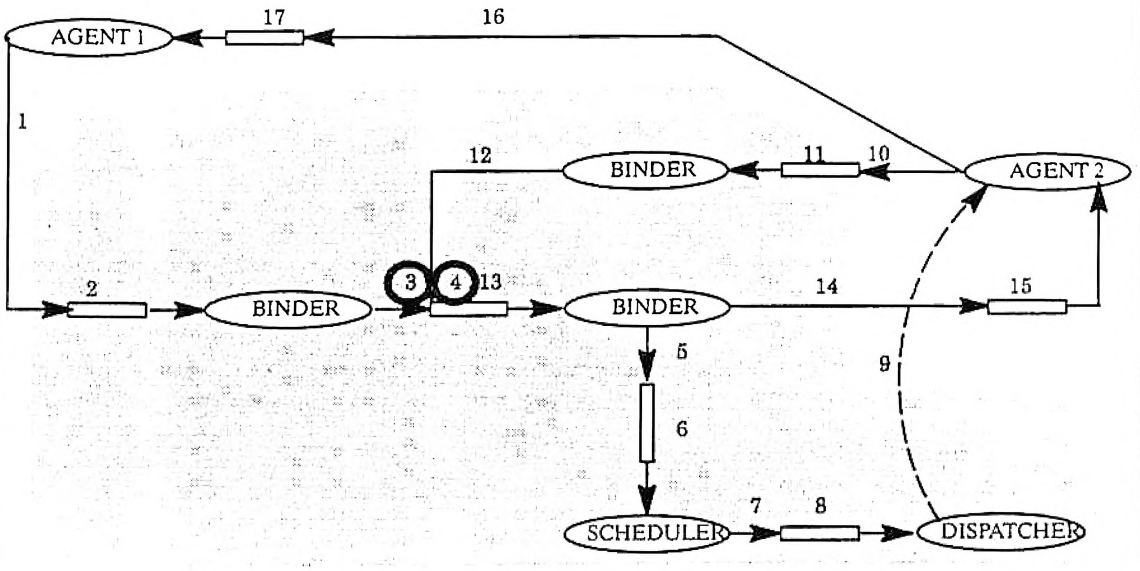

Figure 17. Performance Scenario 6 – Worst Case

2. Analysis. Returning to the conditions presented in table III, when the address of the target agent is not known by the sending agent, one propagation delay and one queue delay are added because the message must go through the binder of the sending agent. When the binder of the sending agent does not register the agents of the desired class, an additional propagation and queue delay are added because the first binder will forward the message to a binder that does track the agents of the desired class. If a binder finds that no agent in the requested class is currently available, many actions must occur.

A message must be sent to the scheduler causing an additional propagation and queue delay. The target agent must initialize, causing an additional delay. Once the target agent initializes, it must register with its binder, adding an additional propagation and queue delay. And, finally, the target agent's binder must forward a registration notice to other binders adding another propagation and queue delay. The time required for the target agent to initialize is dependant upon whether it has already started or not. If an agent has already started, then the average case of 0.5 times the normal initialization time is used. If the initialization has not started, then the scheduler must request it by forwarding a message to the dispatcher. This adds one more propagation delay and one more queue delay. In addition, the full initialization time will be required. Remember that initialization of an application agent involves loading an executable in memory. Table IV shows a summary of this information. In table IV, "p" represents propagation delay, "q" represents queue delay, and "i" represents time to initialize. "Scenario 1" has been shortened to "S1", "Scenario 2" to "S2", and so on. Table IV shows the total time from when the message is sent by the first agent until the response is received.

Table IV. Scenario Summary

Is Address of Target Agent Known?	Does Binder Register Target Agent?	Is a Target Agent Available?	Is a Target Agent Being Initialized?	Total Time
Yes	x	Yes	x	Best: 2p+2q
Yes	x	No	x	Error (see text)
No	Yes	Yes	x	S1: 3p+3q
No	Yes	No	Yes	S3: 5p+5q+$^1/_2$i
No	Yes	No	No	S5: 5p+5q+i
No	No	Yes	x	S2: 4p+4q
No	No	No	Yes	S4: 7p+7q+$^1/_2$i
No	No	No	No	S6: 8p+8q+i

Table IV makes it clear that initializing an agent is considered to be the overriding influence on performance. Scenarios 4 and 5 illustrate this point. Scenario 5 is considered to perform worse than scenario 4, so we have $4p+4q+i > 6p+6q+{}^1/_2i$ which can be reduced to $^1/_2i > 2p+2q$, and further refined to $i > 4p+4q$. The importance of making resources available in advance is thus emphasized.

In comparing the performance of this system to others, remember that even if the worst case scenario is required to establish communication, the best case of direct agent to agent communication follows from that point forward. In addition, because the environment is based upon the software development process, the scheduler component

and the task agents both have the ability to look ahead in order to anticipate requests for agents of a particular class. This capability makes it very likely that an agent will be available when requested as long as the user is working within a process that has the task order and tools required defined. As mentioned previously, the existence of a process driven SEE should not cause an organization to handcuff the developers. The capability must still exist for individuals to use SEE resources that have not been explicitly associated with a task in the process. It would be wise for the SEE to record these events, however, so that the process can be reviewed for modification.

Another point to consider is that an environment that is not process driven does not look ahead to upcoming tasks and prepare the environment since it has no knowledge of upcoming events. Given that the time to load the new agent and associated executable is the largest factor in the scenarios above, the worst case here, which will happen rarely, is roughly equivalent to the common case in current operating systems.

When comparing the performance to current process driven SEEs, both have the capability to look ahead to upcoming tasks so the distinguishing factor comes in the fact that the components introduced here distribute the load of the process engine. In other systems, it is more likely for the process engine to become a bottleneck.

C. SCALABILITY

Task and application agents can be seen as objects. These objects encapsulate either a portion of the software development process, in the case of task agents, or an application tool, in the case of application agents. It is easy to create or revise agents to manage new or modified portions of the software development process. The complexity of adding a new application agent to manage a new application is determined more by the new application than by this collection of SEE components.

When considering projects that may add or subtract a number of users over time, a SEE based upon the components introduced here would simply add or subtract the appropriate number of agents and corresponding binders as required to meet current demands.

IX. MODEL

The previous section mentions that it would be appropriate to use simulation to get an understanding of the rules to implement in the scheduler. This section describes an Ada program that was developed to model the components introduced in the last section.

A. IMPLEMENTATION

The paragraphs below describe the major features and efforts to date related to the program that models the components introduced in the previous section. The purpose of modeling the system is really twofold. First, the model is useful to ensure that the concepts presented in the previous section are sound. That is, implementing the basic operations in the model provides the opportunity to make sure that the concepts are possible to implement. Second, the model is likely to prove useful in developing rules for the scheduler component of the SEE. Ada was chosen to implement the model both because of its tasking mechanism, and because the implementation would provide an interesting educational tool. It was also noted before starting the effort that compilers are beginning to surface that allow Ada tasks to be run on processors across a network (although at fixed locations). This will likely be an interesting application with which to test the capabilities of those compilers.

1. Active Components. Task agents, application agents, binders, schedulers, and dispatchers represent the active components in the system. All active components were looked upon as agents. For example, a binder was viewed as an agent whose class was binder. This is conceptually consistent with the notion of an agent. As a member of the binder class, the binder agents responded using the methods of that class. That is, binders performed binder methods, other agents did not.

For the purposes of validating the concept, it was determined that, apart from the capability to create other agents, there was no need to implement application agent methods, or scheduler and dispatcher methods. This is not to say that these components are unimportant or simple. In fact, the algorithms used by either of these two could spawn a complete analysis of their own. Such an analysis is beyond the scope of this paper, however, and implementation of special methods would only reveal that agents of the scheduler or dispatcher class responded in a manner that was different from agents of other classes. Since the binder and task agent already represented multiple classes, this was already known to be true.

Each agent in the modeled system was actually an independent Ada task instantiated by a main driver routine. Each task had the capability to register and

unregister other agents; to send and receive mail messages to and from other agents; to build event trees, and to rebuild event trees mailed by another agent; and to detect which methods had been triggered by external or internal events and to process the methods in the proper order.

2. **Event Trees.** The software development process was viewed as a hierarchy of subprocesses in this implementation. As mentioned before, a task in a software development process may itself be composed of subprocesses. This concept was simply extended so the process was viewed as one task at the highest level. That task breaks down into subprocesses as the full definition of the process is fleshed out. The software development process itself, then, becomes a hierarchy, or tree.

This approach makes sense conceptually and realistically. Software development organizations rarely work one development effort at a time, and the different development efforts that they work rarely have identical processes. If the process is defined as a single task at the highest level, and all previously stated features are applied, a great deal of power is achieved. Information can now be gathered at the project level in the same manner as at any other level simply by triggering the methods of the task agent assigned to manage the highest level task in the software development process hierarchy.

Because tasks in a software development hierarchy are not only triggered by the completion of a previous task, however, they also require that their entry criteria be met. It makes sense, then to include a "trigger" on every task in the software development process.

The "nodes" of the event tree had the following components:

- a trigger – to define the event(s) that would cause this node to fire and the state in which the agent must be in order for the event to have affect,
- an action – that defined the method that would be executed when the node fires and the state in which the agent will be placed once the method is complete,
- #of lives – defined the number of times this node could fire before it would be deleted, and
- reply – told the agent whether to build a reply message to report results after the action was taken.

In addition, both the trigger and action components included parameters to further refine either the conditions required for the node to be fired, or the action that would take place once it was fired. There was also a component that was unexpected at the

beginning, but proved useful in simplifying the implementation. That component allowed the agent to look past a particular component when it did not fire to see if the nodes below that one would fire. This component had the same structure as the trigger so that it too could specify certain conditions in which the node would essentially disappear if it was not triggered.

Event trees, then, represented the software development process, and were referenced by an agent upon receiving an external or internal event in order to determine when a new action (software development process task) had been triggered. An agent would receive input from another agent, process the input and then look in the event tree to find all triggered actions. These actions were fired in turn and the triggered events collected. This continued until either no more tasks were triggered, or an infinite loop was detected that indicated an erroneous process definition.

3. Communication. Communication among agents was facilitated by building a generic mail package that also used tasks. The address of an agent was actually a "mailbox" task that had been instantiated by the agent. Mail was "smart" in that a message that was sent from an agent was carried by a courier, another Ada task, that knew enough to go to the binder assigned to the sending agent to get a real address when the sending agent did not provide one. If, however, there was some reason that the courier could not reach the binder or the target destination within a reasonable amount of time, as determined by the sending agent, it would return to the sending agent with a flag set that indicated this condition.

This mail package simulates events that occur in real environments and had the built-in feature to use the agent's binder automatically, thus simplifying the statements in the agent itself. In addition, by implementing the communication mechanism in this manner, task rendezvous could be used to determine the number of couriers waiting at a particular mailbox. This number is automatically collected by the mailbox tasks at times when it is not at a rendezvous with another task, for example when a courier is putting mail in the box or an agent is taking mail out. This number is actually a queue size, and can be used to get an indication of the performance that can be expected by the system under various scenarios of agent/binder assignment, different scheduling algorithms, etc [32, 33].

As a final point, mail was instantiated so that the message that was mailed was the same type as a node in an event tree. This provided the flexibility for the sending agent or the main driver to issue a direct command by placing wildcards in the trigger fields, or to simply forward portions of the software development process with the specified triggers and actions in place. In the latter case, the actions represented tasks in the software development process.

B. EXPERIMENTS

Most of the work with the modeling program was actually related to defining the required data structures and appropriately allocating the logic. On many occasions, this work forced the component breakdown whose final configuration is identified in the previous section to be modified. Thus, the modeling program was helpful in defining the system that it modeled even when it was under development.

Naturally, the program was built in parts, with verification of added capability completed with each new piece. Once it was running, verification of the basic functions was achieved. All verifications were visual. To facilitate verification, an "Output" class agent was defined.

The output class agent was the only agent that could write output to the screen. Thus, if any other agent received a message that commanded it to write something to the screen, the method in that agent would instead forward the output to an output class agent for display to the screen.

The first run was to demonstrate that the main driver could start an output agent and direct it to write one message to the screen. The main driver created a message with the "PutMessage" command as its action and sent it to the output agent.

Secondly, the main driver constructed a tree of messages to simulate a software development process hierarchy. It added a "PutSubtree" command as the root of the tree and forwarded the entire tree to the output agent. The output agent then had to reconstruct the tree, recognize that the "PutSubtree" command was triggered and write the subtree to the screen in a textual hierarchy fashion.

The next step was for the main driver to create both a binder and an output class agent, tell the output class agent that it was to register with the binder, build and send the same tree as before to the binder. The binder then recognized that the message was for an output agent, looked in the registry, found the output agent's address and forwarded the tree for output to the screen.

Building the model system and running these experiments proved invaluable in gaining a clear understanding of the system requirements and in verifying that the components introduced in the previous section were based upon sound reasoning.

X. CONCLUSIONS

Goals and service level requirements of operating systems and Software Engineering Environments appear to be converging. Operating systems initially provided basic command interpretation and resource management. SEEs were initially a collection of stand-alone tools. Both now seek to provide object management services and interprocess communication services across a distributed network while providing a single system view to the user.

Goals that are common to distributed operating systems and SEEs emphasize that user interaction with the system should take place in a context with which the user is familiar. They emphasize that specific details of an operation, including performance and reliability mechanisms, should be transparent to the user. They emphasize that the environment should be efficient, reliable, exhibit top performance, and accommodate the addition or removal of environment components.

Current SEEs must provide some services that have features not yet available in distributed operating systems. When these features are added to the services provided by distributed operating systems, a synergy will develop that will likely improve system performance, through non-redundant functional implementation at the proper level, and reliability, through dependence upon a single set of services with proven reliability. Improved user productivity will be the natural result.

Use of a process driven SEE provides not only a high degree of process integration, but also facilitates control, team, and management integration. In addition, a process driven SEE provides a foundation for improved system performance when application software must be loaded or made available in real time.

Components of a SEE have been proposed that will allow integration of the software development process, and provide improved performance over current environments. The components have been modeled in an Ada program that verified the core functions. The Ada program created here provides a foundation for future research of the proposed SEE components themselves, and of Ada compilers.

Defining the software development process is not a simple task. Some portions of the software development process can be automated others can not. Forcing automation where it does not belong could lead to disgruntled users who, once bitten, will hesitate to return to use this powerful approach. Continued research into the proper definition of the software development process is warranted on this account.

REFERENCES

[1] ***The* Rational *Watch***, Summer 1993, Vol. 3, No. 2

[2] **COHESION Environment for CASE: Realizing Competitive Advantages from Software Engineering,** Digital Equipment Corporation 1993

[3] **Capability Maturity Maturity for Software, Version 1.1,** Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, February 1993

[4] **Process Integration in CASE Environments,** P. Mi, W. Scacchi, IEEE Software, March 1992

[5] **CASE Tool Integration: Long-Term Hopes & Near-Term Solutions,** New Sciences Associates, Inc., July, 1990

[6] **Learning from IPSE's Mistakes,** A.W. Brown, J.A. McDermid, IEEE Software, March 1992

[7] **Strategies for Integrating CASE Environments,** M. Jarke, IEEE Software, March 1992

[8] **A Framework for Integrated CASE,** M. Chen, R.J. Norman, IEEE Software, March 1992

[9] **Interacting with the FIELD Environment,** S.P. Reiss, Software – Practice and Experience 20(S1), June 1990

[10] **Integrating Coarse-Grained and Fine-Grained Tool Integration,** W. Harrison, H. Ossher, M. Kavianpour, Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE '92), July 1992

[11] **Modern Operating Systems,** A.S. Tanenbaum, Englewood Cliffs, NJ: Prentice Hall 1992

[12] **Operating System Concepts,** A. Silberschatz, J.L. Peterson, Addison-Wesley, New York, 1988

[13] **The Role of the ISO in Telecommunications and Information Systems Standardization,** E. Loshe, IEEE Communication, January 1985

[14] **Basic Reference Model for Open Systems Interconnection,** ISO 7498, 1983

[15] **Reference Model of Open System Interconnection,** CCITT, Recommendation X.200, June 1984

[16] **Distributed Systems Concepts and Design,** G.F. Coulouris, J. Dolimore, Addison-Wesley, New York, 1989

[17] **Local Area Networks,** G.E. Keiser, McGraw-Hill

[18] **An Introduction to Operating Systems,** H.M. Deitel, Addison-Wesley, 1983

[19] **STSC Software Engineering Environment Report,** Software Technology Support Center, Hill Air Force Base, 1992

[20] **Working Together to Integrate CASE,** R.J. Norman, M. Chen, IEEE Software, March 1992

[21] **Reference Model for Frameworks of Software Engineering Environments,** National Institute of Standards and Technology (NIST) and the European Computer Manufacturers Association (ECMA), NIST Special Publication 500-200, Technical Report ECMA TR/55, 2nd Edition, December 1991

[22] **Definitions of Tool Integration for Environment,** I. Thomas, B.A. Nejmeh, IEEE Software, March 1992

[23] **Tool Integration in Software Engineering Environments,** A.I. Wasserman, Software Engineering Environments: Proceedings of the International Workship on Environments, F. Long, ed., Springer-Verlag, Berlin, 1990

[24] **The CASE Interoperability Message Sets: Release 1.0,** Digital Equipment Corp., Silicon Graphics Inc., Sun Microsystems Incorporated, October 1992

[25] **OSF/Motif Programmer's Guide, Release 1.1,** Open Software Foundation, PTR Prentice Hall, Englewood Cliffs, New Jersey, 1991

[26] **Toward an Object-Oriented Framework for Designing Services in Future Intelligent Networks,** S.J. Greenspan, C.L. McGowan, M.C. Shekaran, IEEE, 1988

[27] **Report of the Defense Science Board Task Force on Military Software,** Office of the Under Secretary of Defense for Acquisition, Washington, D.C., September 1987

[28] **Process Driven Development,** J.W. Over, STARS '92 Conference, "On the Road to Megaprogramming"

[29] **SynerVision for SoftBench – A Process Engine for Teams,** Hewlett Packard, 1992

[30] **Diana Reference Manual,** G.Goos, W.A. Wulf, Department of Computer Science, Carnegie-Mellon University, March 1981

[31] **Frame System Concepts,** Representing Knowledge as Frames, Fundamentals of Artificial Intelligence, McDonnell Douglas AI Center Training Course

[32] **Using Service-Level Indices to Manage the Quality of Computing Services: A Case Study,** A. Roeseler, A. von Mayrhauser, Journal of Systems Software, 1992

[33] **Choosing a Service-Level Indicator: Why Not Queue Length?,** R.F. Barry, J.L. Hellerstein, J. Kolb, P. VanLeer

VITA

John Hayes Lampkin was born on 23 July, 1961 in Decatur, Illinois. He received a Bachelor of Science degree in Computer Science from the University of Illinois at Urbana in January, 1984.

Since graduation, Mr. Lampkin has been employed by McDonnell Douglas Corporation in St. Louis, MO. He played a major role in the development of the Software Engineering Environment that supports the F-15 Operational Flight Programs written in assembly language. He was responsible for developing the Software Engineering Environment that supports the F-15 Operational Flight Programs written in Ada. He currently has a lead position in the McDonnell Douglas Aerospace-East (MDA-East) Center for Software Engineering with primary responsibility for developing environment components for use on all avionics programs in MDA-East.