A KNOWLEDGE ENGINEERING APPROACH TO ACM

by

RANDY G. HAHN

B.S., Kansas State University 1984

A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Major Professor

Table of Contents

List of Figures

ACKNOWLEDGEMENT

To Dr. Elizabeth Unger whose patience, and wisdom helped me complete this difficult task, I am forever in your debt.

Chapter One

1.0 INTRODUCTION

In the past ten years, research in Office Automation has focused on mechanizing office tasks. This includes such tasks as creating word processors, spread sheets, etc. Little attention has been given to automating the functions in an office, for instance automating the procedures that a form goes through to be completed. An office encapsulates many different information concepts, data processing technologies, and communications methods.

There are four distinct problem areas in current OAS research. The first is that of creating a unifying concept for information now found in the office. This problem deals with the intergration of current mechanized systems. The next problem is that of automating the office place, and dealing with unstructured problems. This problem implies that an OAS system must assume more responsibility. The next problem deals with distribution of the system throughout an office. Each node in the network must share common global functions yet retain those functions which make them unique. The last problem area deals with the adaptability of the system to change over time. The system must be flexible enough to expand as the office does. The technology needed to automate an office implies a different kind of knowledge, a higher degree of knowledge than the technology of mechanization.

To solve these four problems found in office automation an integration of current concepts found in Computer Science, AI, and Programming languages is necessary. The primary focus must be on shifting the responsibility off the user onto the system. The responsibility in this model will

fall upon the systems objects. This implies that the objects, which inhabit the system, will have to have enough knowledge to drive the system.

The technology to solve these problems is currently available. The problem is that the concepts needed to create an effective OA environment are spread throughout the Computer Science field. The components which are needed to create the necessary OAS can be found in AI, Programming Languages, Distributed Systems, Object Oriented Programming, and current OAS research. Each of these fields has some piece of knowledge and information to contribute to the development of such an OAS. As of yet, the knowledge in these fields have not been combined to create a system which puts the emphasis on automation. Perhaps one obstacle is the complexity of such a task but the existence of such a system also implies a shift of some responsibility (power) from the user to the system.

In this thesis proposes a system is proposed which is a collection of knowledge integrated in such a way as to shift the responsibility from the user to the system, and is designed to allow an evolutionary movement of knowledge. The responsibility of system execution is placed upon the objects which inhabit the system. Integration of these concepts will create a system where objects have sufficient knowledge to guide there own computation and navigation through a network. The objects will be able to modify themselves, and make decisions based upon their current state. The objects are in essence controlling the office process which a person may now control, i.e., filling out a form, and following the procedures for its completion. The objects will utilize system blackboards, and message passing to communicate with other objects and the outside world as a source of external knowledge and guidance. For a system to achieve this level autonomy it will take the integration of several sources of knowledge.

The field of programming languages will contribute several concepts to the definition of this OAS system. Procedural languages will lend the concept of variable state, i.e., Pascal variable. The concept of evaluation of functions will be drawn from functional programming languages [9, 34]. The concept of a data driven machine will be used from the data flow languages [5]. Concurrent Languages will such as C-Pascal and Concurrent C will lend the concepts of resource allocation, and monitors [32]. Object oriented programming will contribute the concepts of abstract data types, and message passing for the encapsulation of objects and object communication respectively [28, 16, 17, 18]. The form of system control will come from knowledge representation languages which use agendas, and blackboards [31, 13] for system execution.

Concepts from the field of AI have had a large impact on the the design and implementation of this system. The concepts come from the area of AI known as Knowledge Engineering, where the important concept of Knowledge Representation is used. In the last fifteen years researchers in AI have spent a great deal of time studying knowledge representation. They have explored many representations for knowledge, and have produced several conclusions about the representation of knowledge in a computer system. First, the correct representation of knowledge for some problem will facilitate an easier solution to that problem. Second, it is also concluded that real world knowledge comes in two distinct types declarative, and procedural. The proposed model in thesis captures both types of knowledge.

The form of knowledge representation chosen for this thesis is called frames. The frames scheme of knowledge representation was developed by Minsky in 1975 as a theory to explain vision. Frames represent declarative

knowledge well, but are less adept at representing procedural knowledge. Frames are used by this thesis to integrate system knowledge with the representation of the system data and action objects. To capture procedural knowledge this thesis will use a later development called procedural attachments. A procedural attachment can be considered a semantic action attached to some data value or operation. The procedural attachment is executed when some condition defined in the attachment is met [10]. The procedural attachments will be used to facilitate object communication, and modification of system objects. The structured use of procedural attachments will lend itself to a programming paradigm for this OAS model.

The field of object oriented programming also plays an important role in the definition of this OAS. Object oriented programming concepts are derived from the programming language Simula [5]. The first objected oriented programming language was designed by Alan Kay at PARC [28,5]. Kay's language is called Smalltalk. Kay used the class concept from Simula [36] and constructed a language based upon it. In Smalltalk, type definitions are called classes and an instance of a class is an object. A class is an encapsulation of data objects and operations which can be performed upon those objects. The way computation is performed is that objects pass messages back and forth. Classes can be subclasses of a previously defined class, thus hierarchies of classes can be constructed. Objects when they receive a message have an associated method (procedure) which will fulfill a request by another object. If that method is not found in the current object, the object will search up the hierarchy until it locates the appropriate method. These two concepts data abstraction, and message passing will be used extensively throughout the system.

Current object oriented languages will contribute two other concepts -

daemons, and access oriented programming. These are similar in physical representation to procedural attachments but are used differently. The two language which make use of these concept are Flavors, and LOOPS. Daemons as represented in Flavors are associated with some method defined for the object. There are two kinds of daemons in Flavors - passive, and active. The passive daemons react to messages sent to an object. Active daemons can stop messages from activating a method, or alter the message sent to the object. In Flavors the active daemons are called Wrappers and Whoppers [17].

The LOOPS language has a different sort of daemon. The LOOPS daemon is activated whenever the object's value is accessed. These daemons are divided into two groups get, and put daemons. The put daemons are activated when a message attempts to put a value in an object. The get daemon is activated when an object attempts to get a value from the object. These two sorts of access oriented daemons from LOOPS, along with the daemons as defined by the Flavors programming language constitute the attached procedural programming paradigm.

The last object oriented model which is utilized by this thesis is the ACM model defined by Unger [1]. The ACM model is the foundation for the definition of this OAS system. Objects as defined in ACM are extended to reflect the knowledge needed to drive the OAS. ACM gives a basic definition to both data objects and operations. This definition is used to define all objects and operations in the extended ACM system (E-ACM). The other central concept which ACM utilizes is the concept of Petri nets for control. Petri nets capture the concept of data flowing through a network of operations, thus enforcing the idea of an object which is in control of it's own computation. The Petri net concept is utilized as this system's

model for computation.

The concepts from object oriented languages are used extensively in this model, however, the generic definition of this model's data objects and operations are derived from Unger's ACM model. The frame concept will be used to represent the data objects and actions defined in the ACM model. The concepts of data abstraction, and message passing as defined in an object oriented environment are incorporated into the model proposed herein. Daemons are used primarily by this model as a means of communication to implement the attached stimulation and termination conditions of the objects as defined in ACM.

Distributed systems add needed knowledge about networks and process communication. The Petri net concept is utilized in ACM to help control overall system interaction. This technique allows partial ordering, expresses implicit parallelism, and relational dependencies which exist in the system [2]. Two other forms of system control which are used in this system, are the agenda, and the blackboard. The agenda is an ordering mechanism which orders the operations for execution in the system. The agenda controls the priority system which gives each operation a rank, to designate its order in the execution list. The agenda mechanism controls the ready list and the firing list for execution [31]. The blackboard concept is utilized to help facilitate communication in the system. The blackboard will maintain correspondence (stimulation, and termination conditions for objects and operations) for objects and operations [13, 9, 10]. The blackboard enforces mutual exclusion by allowing only access to it by one object at a time. Our system integrates both these concepts to form a powerful system of communication at not only the network level but at the lower levels of computation as well.

A previous implementation of ACM system in a network of CORVIS machines was accomplished by Catherine Carter at the University of Kansas. Carter implemented a simplified version of ACM which did not include the termination condition as part of the object's definitions. Her system is used in this work as a guide and as a comparison system for this extended version of ACM as defined in this thesis.

Several problems found in office automation are amenable to solutions using the extended ACM model represented as frames. In the past ten years office automation has focused on mechanizing the office. A functional approach which results in an integrated system appears to be superior. Automation encompasses mechanization, but also address's the unstructured problems, and procedures found in an office, i.e., processing a form - getting the correct people to work on it, getting management to sign it, and getting the form out on time. Any good OAS should be able to handle such situations. An OAS should not only mechanize the task of a typewriter to a word processor, but also use technology to know when that task should be executed and under what conditions the word processor should be instructed to produce a specific document [22].

Any OAS constructed dealing with automation will have to deal with two problems related to automation[22]:

- As automaticity increases, the control sources moves from an external source to technology.
- As automaticity increases, integration of functions increase.

Office automation systems encompass distribution. A main problem with distributed systems is interprocess communication. The office could be physically spread throughout a floor or and entire building. Each component of the OAS should have shared access to global functions yet each must have

its own set of unique capabilities. The problem is getting all the components to communicate with each other. A related problem to this is that any office has the potential to expand. This includes an office adapting new functions, operations, or the addition of a new office unit to the system. An OAS must be flexible enough to allow the system to expand with the office, the OAS should reflect the evolution of the office place.

The extended ACM model allows both objects - data objects and operations, to have stimulation and termination conditions. These conditions will initiate and terminate actions in the systems. Objects now have the ability to set in motion the process of computation, thus facilitating a form of automation. For instance in filling out a form, the data object (the form) will know when to initiate its process (stimulation condition), where in the network it must go, what users are allowed to access it, and when it must be stopped (termination condition).

This model is designed to solve some of the problems found in OAS today. The concepts used to create this model are current ones found in several fields of Computer Science. This thesis discusses a method for the implementation of the termination and stimulation condition as defined by the Unger model. The system will integrate an agenda and a blackboard mechanism to control computation, and process communication. The object oriented concepts of data abstraction, message passing, multiple inheritance, and daemons are used extensively throughout the system. The objects are represented physically by the frame knowledge representation scheme, and treated conceptually as an independent unit. This unit is an encapsulation of objects and operations, and knowledge (see Figure 1.1.1).

The frame representation technique enforces the concept of abstract data typing found in object oriented programming, and the ACM model.

Frame theory also allows for the use of procedural attachments. These attachments and the daemons as defined by object oriented programming, are used to define a programming paradigm. The goal of this thesis is to capture the knowledge necessary to create an intelligent data object, which will function in the office environment. This thesis will also discuss a partial implementation of the E-ACM model.

Summary of E-ACM Concepts

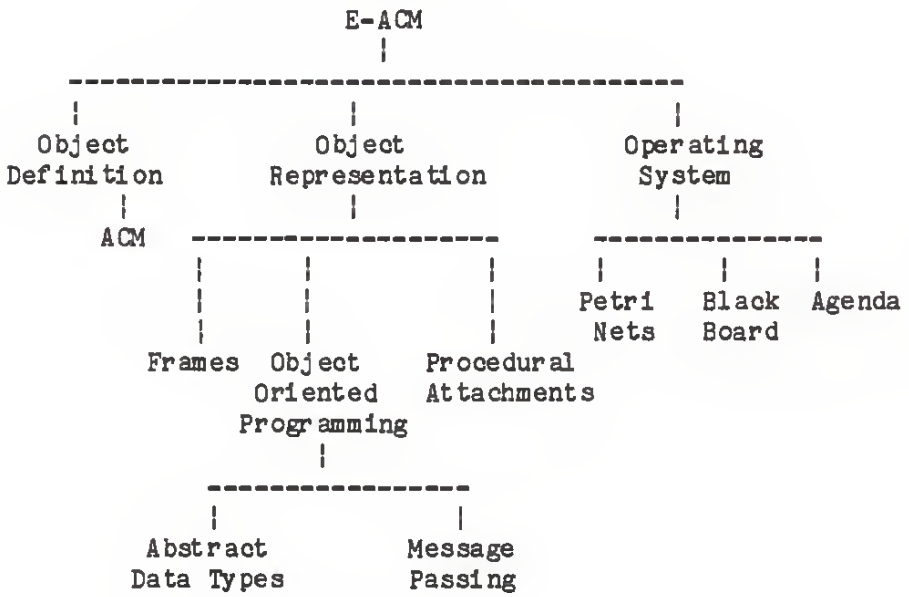

Figure 1.1.1

The discussion of the implementation will cover the problems encountered, and the extension that the implemented system needs to be a complete E-ACM model (OAS).

The following is a break down, chapter by chapter, of the contents of the rest of the thesis:

Chapter 2.0 : This chapter is divided into two sections. The first section is a review of the original ACM as defined by Professor Unger. The second section is a review of an Implementation of the ACM as done by Catherine Carter.

Chapter 3.0 : This chapter defines the representation concepts for E-ACM, it is divided into three sections. The first discusses Knowledge Representation and in particular Frame systems. The second section discusses object oriented concepts. The third section discusses the use of procedural attachments in this model.

Chapter 4.0 : This chapter defines the E-ACM model. It defines the Data Objects, Actions, and Operating System.

Chapter 5.0 : This chapter discusses a partial implementation of the E-ACM model as implemented in the Pearl frame system.

Chapter 6.0 : This chapter is divided into three sections. The first gives a summary of this research. The second provides some conclusions about the E-ACM model. The last section discusses future work for this model.

Chapter Two

2.0 BACKGROUND for E-ACM

The ACM model was first proposed by Professor Beth Unger in her dissertation [1]. The model was designed to fulfill two goals: (1) represent information in one encompassing structure, and (2) encourage structured program development [1]. ACM was conceived in such a way that the data objects provide useful and essential properties needed to operate in a concurrent environment. The purpose of this Chapter is to review the background and original concepts which lead up to the design of the E-ACM system. This Chapter is divided into two sections. The first is devoted to a summary of Unger's original data object and action definition. The second section is a review of the ACM system as implemented by Catherine Carter at the University of Kansas [4].

2.1 A Concurrent Model

The initial model concept for a concurrent environment, called ACM, was defined by Professor Unger in the dissertation. This section will review her theory, and define the model's objects and operation. The theory is built around two concepts: data-flow, and Petri nets.

ACM was designed around the control concept of Petri nets. Petri net theory allows objects to flow through a bipartite network of nodes and transitions. The transition in the net represents actions which operates on its input. The operation then generates output. Figure 2.1.1 provides an example Petri net:

A Simple Transition Petri Net

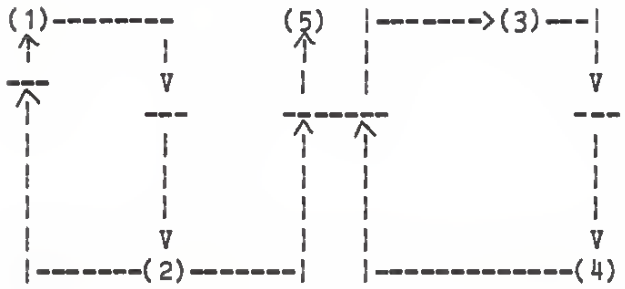

Figure 2.1.1

Conceptually, Petri nets allow for concurrent execution, with the restriction that the transitions only fire if their input arcs each contain a token [2]. Petri nets are also characteristically non-deterministic. For more on Petri net theory see [2, 3].

The second concept used in the design of ACM is the idea of data drive from data flow theory. Data drive contributed two characteristics. The first is that of a data driven machine. This means that the data flows through some set of operations (transitions). The second characteristic is the data flow concept of the single assignment rule. This rule restricts the value of a data object to just one value during the execution of a program. Another definition given for the single assignment rule is that an object may only appear once on the left hand side during the execution of a program [5]. The characteristic that the data flow model shares with Petri nets is that a data flow system exposes inherent parallelism. Here is an example data flow graph [6]:

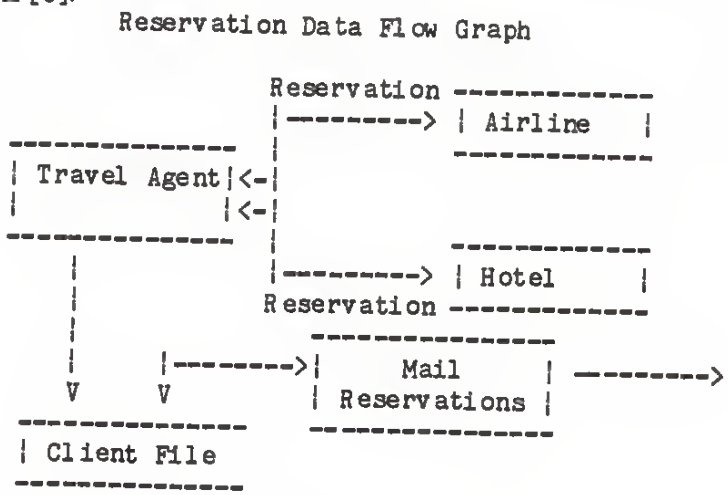

Figure 2.1.2

The data flows into the travel agent, and from there it flows through the rest of the system. This data flow graph captures the inherent concurrency in the system. The second attribute which the data flow model and Petri nets share is that both are very modular in structure.

The ACM model has two primary actors: data-objects and actions. The system is uniform in that every object despite its complexity is defined from the same meta-object. Actions are defined in the same way. A meta-action definition is given from which all actions are defined. The functions can be extremely complex but the ACM model is flexible enough to define any operation completely. The first actor to be reviewed will be the data-object, since this is the active entity (moves from operation to operation) in the system.

2.1.1 Data Object

A data object in ACM is defined as a seven tuple. The Object consists of a Designator, an Attribute, a Representation, a Corporality, a Value, a Stimulation condition, and Termination condition. Collection of these tuples together compose any object defined in an ACM environment. These characteristics allow any data object of any complexity to operate effectively in a concurrent environment.

2.1.1.1 Designator

The Designator of an object consist of a four tuple. This tuple is an environment context, user name, instance name, and an alias. The context component is used as an index to a data object if it is operating in a different environment. Context names are generated as an object moves from one environment to the next. These names are concatenated as the object moves to outer environments. As the data object moves from the outer environment back to the original environment the names are unconcatenated respectively.

The user name is simply a name given to the object when it is created. This is done when the program is defined.

The instance component is used to keep track of replicated objects. The system must assign to each duplicate an individual name to maintain the integrity of the system, and reduce the possibility of ambiguity. The instance component of an object consists of a three tuple: (s,t,o). Where

s is the name or notation for spatial position.

t is the chronological identity generated by a user and specified by a clock. The clock produces monotonically increasing sequences of values.

o is an integer sequence number that is generated starting with an initial value, specified by the user.

The last component of the Designator is the alias. This tuple is only used when the object needs to be referenced in another context. The syntax Designator tuple is:

<Designator> ::= <Context> .. <User Name> .. <Instance> .. <Alias>

An example of an Object's Designator is:

machinea .. budget .. 1980 .. 10

The first name corresponds to the context the data object resides in: name = machinea. The second name represents the name given by the user: budget. The third tuple represented is the instance tuple, 1980, and the last component, alias, equals 10. The alias represents how this data object is referenced outside the current environment.

2.1.1.2 Attribute

The Attribute component of an Object tuple is defined by three components: Object type, physical data representation, and relationships the object has with other objects and actions. The first attribute, type, can be broken into two classes: simple and complex. The simple types are integers, reals, booleans, and characters. The complex structures correspond to arrays, records, files, and combinations of the above. The physical data representation is restricted by the hardware used to implement the ACM model. The last component of Attribute is the relationships it holds with other objects in the system. These other objects are either Actions or other Data Objects.

2.1.1.3 Representation

Representation is the third component used to define an Object. This component is defined by three attributes: Object location, coding scheme, and packing considerations. The user is not allowed to manipulate the representation component of the data Object, giving the Object a data-abstraction quality. The location component of the Representation tuple is the physical location of the Object in the system. The coding scheme correlates to the type of character set being used to represent the objects, i.e., 7-bit ASCII. The last component of the Representation is packing consideration. The packing information is used to transmit an object from one environment to another. This allows an environment to transform an object into the correct representation, so that the object can function in the next environment properly.

2.1.1.4 Corporality

The fourth tuple component is Corporality. This component is defined by four components: longevity, location, replication, and authorization. The longevity attribute expresses the durability of an object in a given environment. This component allows the Object to exist in one of four states: fixed, static, dynamic, or fluid. A fixed Object receives its value during the inception of the model (program). This state is similar to the constant declaration in a standard Von Neumann Language. Static Objects receive their values once during the execution on the model. This type of Object value may not change again during the existence of the model. Objects of this state are similar to the data flow concept of single assignment. The dynamic Object is defined by the system and is history sensitive. This type allows an Object's value to change the name by a temporal or sequential identification, and all past values must be recorded. The last state is fluid. A fluid Object can change in value but its past values aren't recorded. The fluid state is similar in purpose to the standard concept of a variable in traditional languages such as Pascal, PL/C and Fortran. These four states - fixed, static, dynamic, and fluid - define the longevity component of Corporality.

The second attribute of Corporality is location; where the object can be found. The Replication attribute stores the number of replications that have been made of the object. The last tuple is authorization. This tuple restricts the kinds of operations which are allowed work upon the object. Thus the object has another similarity to an abstract data type in that only operations specified by the object may be allowed to operate on it. The four tuple - longevity, location, replication, and authorization - composes the Corporality characteristic.

2.1.1.5 Value

The fifth component of an Object is its Value. The Value component represents the informational unit which the object was created to represent. Value types come in two forms - simple atomic values and aggregations. A simple atomic value is an integer, real, boolean, or character. Other simple atomic values are sets, ordered collections, and actions. These simple types are embedded in the system.

The aggregation type is the data abstraction facility in ACM. An aggregation allows the user to define a structure composed of one or more objects. Once composed, the basic operations which can be applied to the structure must also be defined in the object. A structure is composed of a three tuple: a structure, a set of partitions, and a set of legal actions where [1]:

structure - is a structure (designator, attribute,
representation, corporality, value) composed
of n objects oi = (designatori , attributei,
representation, corporality, value),
1 <= i <= n.

partition - is a set of subcollections or partitions which
can be referenced as a structure.

legal actions - is a set of legal action which can be
preformed on the structure.

The one restriction placed on the user employing such an aggregation is that he/she does not have access to the underlying data object representation.

The restriction on giving an Object a Value is that a Value given to an object can only be defined after the Designator is defined.

2.1.1.6 Stimulation and Termination

The last two components of an Object are the Stimulation and the Termination conditions. The Stimulation component is used to activate an inactive Object. This may mean moving the object from one environment to the next, or that its value is now needed for some operation to be formed. The Termination component signals an object cease to exist in the system or in an operation.

2.1.1.7 Object Summary

The Data Object in ACM is composed of a seven tuple: Designator, Attribute, Representation, Corporality, Value, Stimulation, and Termination. All data objects must be defined by the meta-object just described. Some of the components which make up the data object are allowed to be empty. A conceptual graph of the whole object may be found in Appendix One. This data object is very flexible, and uniform in its treatment of data representation. It is also analogous to an abstract data type defined by [5].

2.1.2 Action

The Action definition is the form operations take in the ACM model. Actions are needed by every language to accomplish some task. Action in the ACM system is represented by the five tuple: Stimulation, Material, Action, Request, and Termination. The Stimulation and Termination conditions in the Action object are similar to the Stimulation and Termination tuples in the Data Object. The Stimulation tuple is simply a conditional expression used to trigger the Action to perform its function and operate on the data objects present in the system. When a termination condition is met, it causes the Action to cease execution. A Termination condition may

also cause other actions to cease execution, or stop other actions from triggering execution.

The second tuple, Material, is a list of necessary data objects which the Action will operate on. These data objects are similar to the in/out variables in an Ada procedure header.

The third tuple which defines the form of an object is Action. The Action component is the actual definition of the operation which performs on the data object(s). For example, this could be a simple add operation, or a complex sort operation.

The fourth component of the Action definition is Request. The Request tuple is a list of requests the Action spawns either after it executes or before it can execute.

The five tuple - Stimulation, Material, Action, Request, and Termination - defines the meta-object for an ACM Action. This means that any Action defined in the model will have these components. Not all the Actions will have values for all the components. For example, the Stimulation and Termination conditions may be null. The full definition of an Action Object can be found in Appendix One.

2.1.3 ACM Control Constructs

The ACM model contains objects which can be uniquely identified, and operations which perform some action on those objects. This section looks at flow of control, and how constructs are created. A construct in ACM is a "systematically organized set of requests used to control the flow of a computation." [1] Constructs operate in three ways [1]:

i) Support conditional computations on any basis.
ii) Support alternation computations based on context. iii) Allow for partial computation on incomplete material.

The conditional computation is represented by the Stimulation condition of the request. The conditions allow the system to choose between two actions with stimulated conditions, indicating that the system allows for the concept of alternation. The action's stimulus condition, however, must be mutually exclusive. Here is an example of the alternation condition in ACM:

An ACM Alternation Action

[type = 'MEDICINE'] : MEDTAX (total ; sale)
[type = Not 'MEDICINE'] : TAX (total ; sale, 1)

Figure 2.1.3

The ACM system extends the alternation construct by allowing caseation. Caseation is a construct which provides for the selection of one and only one path from a set of computational paths [1]. The caseation in ACM relies on the stimulation condition of a data object and action just as the alternation construct does. This ensures consistency and the integrity of the system.

The ACM model not only allows forms of decision structures but also repetition. Repetition in ACM takes two forms: repetition and iteration. Repetition is defined as a construct consisting of one request (Si,Mi,Ai,Ri,Ti) where there exists one or more objects in the component Mi [1]. The iteration construct "is expressed as a single request in which the termination condition is present." [1]

The last form of control constructs which will be touched on is that of partial computation. The computation continues until a missing object of material is needed.

2.1.4 ACM Summary

This concludes the review of the ACM theory. In this section the impact of Petri net and data flow theories on the ACM model was discussed. The definitions for both the Data and Action Objects were given. The Data Object is defined as a seven tuple - Designator, Attribute, Representation, Corporality, Value, Stimulation, and Termination. The Action is defined by a five tuple - Stimulation, Material, Action, Request, and Termination. This section also covered the key concepts in control constructs - alternation, caseation, repetition, iteration, and partial computation. For a more in-depth view of the total ACM theory, the reader is directed to dissertation Professor Elizabeth Unger's [1].

2.2 An Implementation of ACM

This section is devoted to a review of the version of the ACM model implemented by Catherine Carter at the University of Kansas [4]. The following three sections will discuss - the Corvus Machine, the operating system she designed for her thesis, and how she represented her Actions and Data objects.

2.2.1 ACM on a Network of Personal Workstations

The ACM model Carter implemented, like the ACM theory, used two Data Flow concepts. The first, the data driven machine meant that the system implemented was driven by the data introduced into it [5]. The second concept, the single assignment rule, restricted all data objects to be either Static or Fixed (according to the ACM theory). Carter's system was con-

structed of six parts:

1) Operating System - resource management.
2) Editor - for program entry.
3) Display - for monitoring the progress of an executing program.
4) Interface - for conventional programs.
5) Communication - facility among the network of Computers.
6) Ability to implement concurrent processing.

The major restriction imposed upon the implementation was that no master/slave relationship could exist. The control mechanism used in this implementation was called a Task Tree.

The Task Tree represents conceptually and physically the different levels of computation in the system. At the highest level of the Task Tree is the abstract concept, and the next level down is used to represent all individual processors. When a user signs onto the system a Task Tree is created for that user on a certain processor. As the user generates requests the system in essence hangs each request on the tree. This makes each request a new branch in the tree. All requests have a path back to the root node. This tree represents the execution data structure, the execution of the tree nodes being done in an in-order manner.

Carter's system utilized the token storage concept, which is used in current data flow machines [5]. This storage technique allows for maintenance of the operand values. There are three trade-offs in by using this

storage technique [4]:

1) It requires more memory space for each request.
2) Less firmware is needed since a separate matching unit is not requested.
3) It fits the model being used.

These last two paragraphs outline the basic principle of the Carter implementation. The next two sections give an in-depth view of the Operating system, and internal representation of system objects.

2.2.2 Operating System

Catherine Carter's operating system is similar to the standard definition given for an Operating system. Her OS fulfills four needs of the ACM model:

i) Scheduler - Both long and short term.
ii) Load Balancer - Distributes work load.
iii) Command Interpreter
iv) Message handler for communication between processors.

The long and short term scheduling dictates when a process can execute, or where a request shall be placed on the Task Tree. The long term scheduler is responsible for activating stimulated Actions, and scheduling when the actions will fire. A dispatcher, which is associated with the Scheduler, selects which of the triggered Actions will fire. This decision process takes place when the actions are passed to the processor for execution.

The structure of the system is a Task Tree, and as such it could become an unbalanced Tree. The main function of the load balancer is to detect an overloaded processor. The balancer also decides which Action will be handled by which processor. It selects nodes in the Tree which will be sent to other processors in the network. The system can balance a Task Tree

because each action is labeled with a tag. This tag marks which program, location in the task Tree, and processor the action is attached to. This allows a foreign task to executed by another processor, and allowing the processor to check the tag and send the action back to the appropriate processor with the appropriate results. The balancing of a Task Tree takes place periodically, as timed by the OS.

In the ACM theory the designator component of the data object must receive a unique name; in Carter's system this is done by the OS. When a user signs onto the system, the OS gives that user a unique i.d.. The i.d. is used as a preface to all objects and operations in the user environment. This naming mechanism is used by the graphic package to illustrate the execution path of a Tree.

The next important component of the Carter implementation is the representation of the objects and operations.

2.2.3 Object and Operation Representation

The implementation language chosen was Pascal. Pascal was chosen because on the Corvus, it has a data abstraction facility. Another reason Pascal is chosen was because it automatically checks type. The ASM68K language was used for reading the keyboard and converting bytes to real numbers during transmission between processors.

The representation of the Task Tree consists of three logical types of nodes. The first node is the user's node. It includes the user's name and the amount of work he/she needs done. The second node is called the interior node and is primarily a place holder. It contains requests for actions, and it passes values up and down the Tree. The last node is the terminal node,

which can't be further refined (i.e., arithmetic operations). These three nodes all share the same representation, but have different functions in the system.

The representation scheme used to represent nodes (actions) and data objects is the record. This structure depicts two concepts well. The first is that a record represents a conceptual unit or packet of information. This allows each packet to be equated with individual processes. The record maintains the packet state whether it is enabled or disenabled. Thus, if this packet has to be sent to another process, everything it needs can be packed up and sent to the other processor as one physical unit.

The second concept the record structure handles nicely is that tokens can be easily stored as subrecords in actions. These substructures can indicate if the material needed for execution has been received or not. Each piece of material (needed data objects) and request name is assigned to a separate subrecord. The record data structure, a Pascal record, is the mechanism used to represent the action, data objects, and nodes of the Task Tree.

Carter made three modifications to the original ACM model in the representation of the action and data object. The first modification was an extension of the Action definition to include a Path tuple. This tuple is used to keep a record of the path from the current node to the root node. The record is used by the system to transfer data objects up the Tree. The second modification was to make the Data object was a subrecord not an individual entity. The third modification was that the Termination tuple for both the data object and the action Thus, Carter implemented a subset of the original ACM model.

2.2.4 Implementation Summary

This section reviewed one implementation of the ACM model as done by Catherine Carter at KU. Her system used two data flow concepts - a data driven machine, and the single assignment rule. The operating system in her version used a token match scheme to match tokens to operations. This system could balance one processor's Tree by shifting the work load to other processors. The basic representation used in Cater's system was a Task Tree consisting of Pascal records. The record fulfilled two purposes: 1) It treated all actions as conceptual processes, and 2) objects could be stored as subrecords in the action record. In the implementation ACM was modified in three ways. The action definition was extended to include a Path tuple, the data object was embedded inside an action, and the Termination tuple wasn't implemented on either the action or the data object.

2.3 Summary

The main function of this Chapter is to give the background material for the ACM concept designed by Professor Elizabeth Unger. This material covers both the original concepts and one successful implementation. The ACM model objects are the data objects and the actions which operate on the data objects. The definition for these objects gives ACM a data abstraction facility. The constructs inherent in ACM are alternation, caseation, repetition, iteration, and partial computation.

The second section illustrates that the ACM model is a feasible system. An implementation by Catherine Carter at KU is currently being used by the CS department there for the undergraduate operating systems course. This section also gives some characteristics of ACM which were modified

and not implemented. For instance the Action definition of ACM was altered to include a Path tuple. This tuple is used to keep a list of its ancestors. Also, the data objects were embedded inside the actions, instead of treating them as separate entities. The implementation didn't include the Termination tuple in either the action or the data object.

This paper then poses the question: What effect can knowledge have upon the ACM model?

Chapter Three

3.0 KNOWLEDGE REPRESENTATION FOR E-ACM

"All men by nature desire to know"
ARISTOTLE, METAPHYSICS

What is meant by the phrase "to know"? Does it mean that people desire to know facts about the world or that they want to know how to accomplish some task, or both? Computer Science, and A.I. in particular, have come to realize over the past 25 years that how a problem is represented can make that problem easier or more difficult to solve. If a problem is poorly represented, finding the solution to that problem will be difficult, if not impossible. Consequently finding the correct representation for a given problem is itself the major problem.

In the past 15 years a subfield of Artificial Intelligence called Knowledge Engineering has struggled to find one general-purpose form of knowledge representation. Knowledge Engineering has produced at least five forms of knowledge representation - Semantic Network, Rule-Based system, Logic, Conceptual Modeling, and Frame Theory. However, it is the conclusion of this research that there currently is no one general purpose form of knowledge representation [10].

Knowledge is power and how that knowledge is represented in a computer system is paramount. This chapter covers three topics of vital importance to the E-ACM model. These topics are knowledge representation, object oriented concepts, and programming with procedural attachments.

3.1 Knowledge

Knowledge about the world can be divided into two types- declarative and procedural. Declarative knowledge is simple fact, discernible in the real world. Declarative knowledge is "knowing that" about something. Procedural knowledge, on the other hand, isn't a simple fact but rather a collection of heuristics which, when integrated together in a certain way, represent working knowledge. Procedural Knowledge is generally called "knowing how" to do something.

This chapter will define declarative knowledge as knowing that, and procedural knowledge as knowing how. To further illustrate the difference between the two types of knowledge consider, the following. An example of "knowing that" is when someone knows for a fact that Kansas State University, in the year 1985, lost all but one of its football games. This sad fact can be verified by looking up the KSU football records for 1985, and therefore is declarative knowledge. Procedural knowledge is "knowing how" to do something, and is harder to represent in a system than declarative knowledge. For an example of procedural knowledge, consider how someone swims. Then think about trying to teach someone to swim. The person wanting to learn how to swim could go to the library and read a book on swimming, but that wouldn't mean the person knows how to swim. It would only teach that person about how to swim. Knowing about and knowing how are two different things. To know how to swim the person would need to take swimming lessons, or perhaps try the sink or swim method.

Artificial Intelligence, and in particular Knowledge Engineering, has explored and experimented with many forms of knowledge representation. The topic of knowledge representation has evolved over the last 15 years

into one of the biggest issues in A.I. It is an accepted fact now that the correct knowledge representation will facilitate the success of a system [35]. Consequently, There have been several knowledge representation schemes generated over the last 15 years. These systems are very good at representing declarative types of knowledge. The weak point in all of them, though, is that they do not represent procedural knowledge very well.

3.1.1 Knowledge Representation

It has been determined that the right representation of knowledge for a given system will allow that system to function more efficiently [10, 35]. If the chosen representation of knowledge is poor, the system is doomed to failure [35].

Researchers have created several types of knowledge representation models. These models can be broken down into five categories [7]:

- Semantic Networks
- Logic
- Rule-Based systems
- Frames
- Conceptual Modeling

The representation scheme which is utilized for the E-ACM system is the Frame representation concept. For further reading on the other four types of representation schemes, the reader is directed to [8] for a Semantic Network discussion, [9] for a review of Logic representations, see Newell and Simons' book entitled "Human Problem Solving", 1972 for Rule-Based systems, and [14] for Conceptual Modeling. For a survey of all the current forms of knowledge representation, the reader is directed to [10, 13]. The rest of this section will be devoted to an in-depth look at Frame theory.

3.1.1.1 FRAMES

There is abundant psychological evidence that people use a large, well-coordinated body of knowledge from previous experience to interpret new situations in their everyday cognitive activity [Bartlet 1932, 10]. With this in mind, Minsky (1975) [10] proposed a model called Frames. This, he theorized, was the basis for holding the information necessary to understand visual perception, natural language dialogue, and other cognitive behavior.

According to Patrick Winston of MIT, a "Frame is a collection of semantic net nodes and slots that together describe a stereotypical object, act, or event" [9]. This means that a frame collects all the relationships present in a semantic node, and treats them as slots in a frame structure.

A Frame consists of four components: designator, slots, fillers, and facets. A definition for each component is given as follows [7]:

Designator : A conceptual name given to uniquely identify a Frame.
Slot : Attribute of the concept given in the Frame.
Filler : The value contained in the slot component.
Facet : The restriction on the type of filler for a given slot.

Conceptually, a Frame can be considered a box with the following form:

An Example Frame Definition

Designator		
Slot	Facet	Filler
Slot	Facet	Filler
" "	" "	" "
Slot	Facet	Filler

Figure 3.1.1

A Frame allows inheritance, deamons, default values, and perspectives [see section 3.1.1.2]. When Frame representations are designed, they usually have the characteristic of being hierarchical in structure. Thus a Frame is either a subclass, superclass, or a terminal node in a tree-like structure. The links between frames in the tree-like structure facilitate inheritance of superclass properties to subclasses. When designing a system with a hierarchical structure it is important to push the knowledge as high up in the hierarchy as possible. This type of design allows the knowledge in the tree to be accessible by more subclasses via the inheritance mechanism.

There are four types of Frames [7, 14]:

1) Class Frames - this type of frame is a template frame which describes a class of information.
2) Individual of a class - This frame is an instance of a class frame.
3) Perspective Frame - This frame defines the context for how a class of frames will be viewed.
4) Prototype frame - Specialized concepts in one or more schemata to show the form of a typical individual. Unlike aggregation, a prototype specifies defaults that are true of a typical case, but not necessarily of any particular case.

The following is an example of converting a representation from a semantic network to a frame model [9]:

A simple Semantic Net

(___) -> denotes a concept

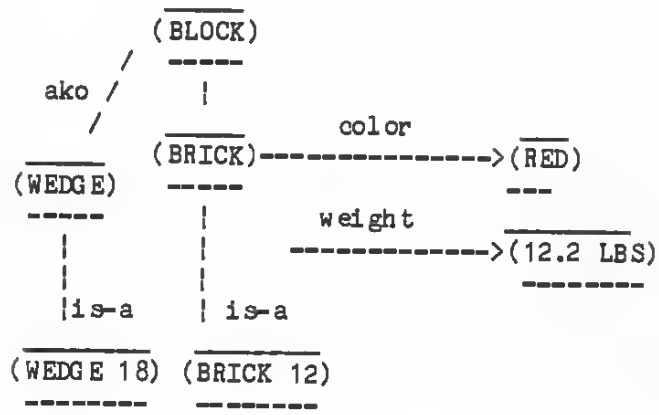

Figure 3.1.2

A frame system

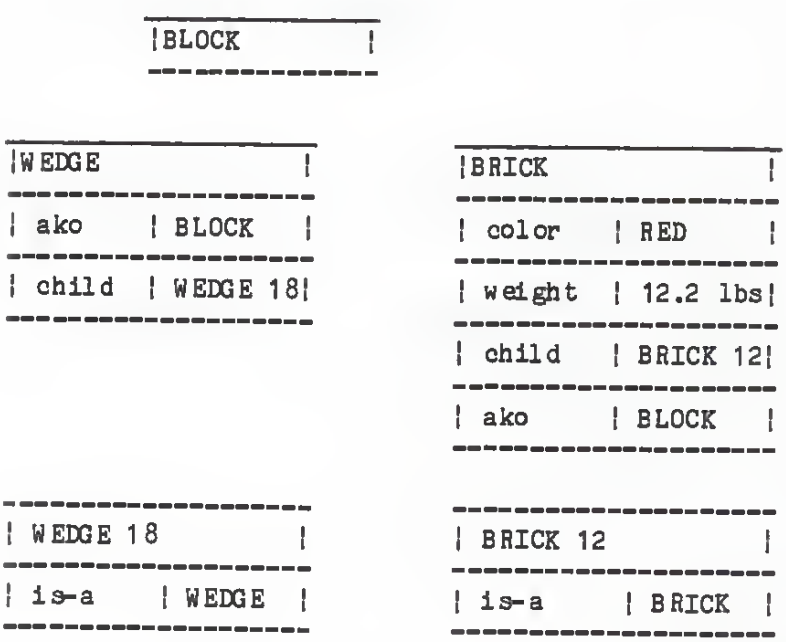

Figure 3.1.3

The frame system reduces the number of links in a semantic network, while still retaining the meaning of the original structure. A frame can then be treated as a conceptual unit instead of as a semantic node with its associated links. In keeping with the concept of inheritance, frames are specializations of Semantic networks.

3.1.1.2 Inheritance, Daemons, Defaults, Perspectives.

Frames utilize the concepts of inheritance, daemons, defaults, and perspectives. This section will define the context of these concepts with respect to the frame representation scheme.

Inheritance

The concept of inheritance was first introduced by Aristotle in his theory of categories and syllogism [14]. He constructed a type hierarchy to represent many things known to man. Every class type in the hierarchy was derived from a set of ten primitive types, such as fire and water. He defined new types by the method of genus and differentia utilizing the ten primitive types. He used the method of syllogism for analyzing the inheritance of properties.

Inheritance allows description movement from classes to instances and subclasses [9]. It should be noted that without a hierarchy, this concept could not exist. Inheritance is demonstrated by the way humans assert things about something once its identity is known. For example, if one sees a robin sitting in a tree, the observer can assert many things about that robin. The observer knows the robin is an instance of the general class of birds. The robin inherits many of the properties (characteristics) from the

class of birds, such as the physical attribute of wings and the ability to fly. The robin, it is said, inherits the properties of wings and the ability to fly from the general class called birds. The only time people fail to assume a correct inheritance property is when an exception is being observed, e.g., a penguin.

Daemons

Daemons are also called attached procedures, active values, hooks, if-added and if-needed procedures [9, 13, 15]. Daemons were first introduced by a physicist named Maxwell, and are sometimes called Maxwell's daemons [7]. Maxwell used the idea of a deamon to illustrate a point about pressure equilibrium in a gas system. Graphically the gas system looks like this :

A Gas Equilibrium system

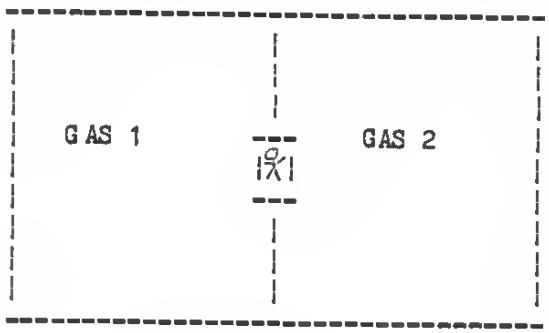

Figure 3.1.4

Inside this system was a little fellow who could alter the pressure in either chamber. He was called a deamon. This deamon controls the gate between the two chambers and if he wanted he could open the gate, and pluck atoms out of one chamber and put them in the other. This theory is contrary to the the gas laws. The gas laws state that when such a gate is opened the

pressure in the two chambers will after a time reach equilibrium. The main point of the deamon was to illustrate that something, unseen to the user, could monitor the current state of a system. It could also has the ability to alter values if it so desires.

In the field of Artificial Intelligence daemons are used for a similar sort of job. They monitor the state of a computation, and by various means can cause actions to take place that effect the state of a computation. In frames these are either called if-needed procedures or hooks. In object oriented programming they are called daemons, wrappers, whoppers, or active values [16, 17, 18]. In section 3.3, a full description will be given about the impact daemons have on the E-ACM model.

DEFAULTS

Default reasoning is used in computer systems when there is an absence of information about something, and no daemons exist. When no information is available it is accepted practice to make a sensible guess, as long as there is no evidence to contradict it [19]. This type of reasoning is called nonmonotonic. Monotonic reasoning states that if a conclusion is derivable from a certain collection of facts, the same conclusion remains derivable if more facts are added [10]. Nonmonotonic, or default reasoning, states that the addition of one piece of information may force the deletion of another [19].

Default reasoning is used in the construction of a knowledge base to fill in the gaps of absent knowledge. The values used for default values are defined as general case knowledge, and most the probable choice. For instance, here are some default examples:

- Most people like flowers.
- Most dogs have tails.
- The most common hair color for a Swede is blond.

The reader should make two observations about these assertions: 1) The default values are for classes of objects, and 2) these assertions are based on commonly believed facts. The important concept to remember is that these assertions are made in the absence of complete information.

PERSPECTIVE

An object can play many roles, depending upon how that object is viewed. Each role is called a perspective. A racing bicycle may be fast for a bicycle but slow as a means of transportation. Perspective deals with what context an object is in, or can be in. A person can be a parent, an instructor, and also a professional bowler, and depending upon the context he appears in, that person can be viewed differently [7].

A perspective is represented in an A.I. system in something a bundle. The object or concept which can be viewed a certain way is associated with a particular bundle, each bundle representing a different perspective. One object can be connected to several different bundles, as was illustrated by the previous example of a person. An object may have several perspectives that may overlap, or even be in conflict. For a more in depth view of perspectives see [9].

3.2 Object Oriented Concepts

The frame representation scheme is integrated with object-oriented concepts to create a powerful representational tool. The central concepts of

object oriented programming originated from the programming language Simula [36], the fundamental being those of objects and classes [28]. The Simula programming language, however, uses the standard data/procedure-oriented approach for communication and control. The first object oriented language was designed at the Xerox Learning Research Group by Alan Kay [5, 28]. The system Kay developed is called Smalltalk. The Smalltalk system is composed of objects which interact by sending and receiving messages. The sending and receiving of messages cause computation to be performed in the system. Flavors and LOOPS are currently two of the more powerful object oriented languages to arise from the concepts established by Smalltalk [17, 18].

In object oriented languages, class represents data type and data objects are instances of the class. The operations are defined inside the class and are referred to as methods. Methods have two functions - to respond to a message by operating on the values defined in the class, or te send messages to other objects [30]. The methods defined by the class constitute the class protocol. This protocol defines the responses of a class to the type of messages received.

The two concepts object oriented language concepts this work utilizes are abstract data types and message passing. The data abstraction facility used by Smalltalk, Flavors, and LOOPS is captured by the class definition. An object class is an encapsulation of data objects and operations, and is captured by the E-ACM system through the use of frames to represent objects. The standard frame representation is extended to include slots which contain executable operations. These operations, in effect, are equivalent to the object-oriented concept of methods. The slots contain the actual code which will respond to messages, or perform some computation.

These methods also can have procedural attachments, which will help facilitate the methods' function (see section 3.3 on procedural attachments).

The primary purpose of the methods is to facilitate a message passing mechanism in E-ACM, creating an interprocess communication facility. Message passing supports the concept of abstract data types, because an object is considered a distinct entity which can communicate with other objects [30]. The only way to communicate with an entity is to send it a message and prompt it for a response. A simple example is the termination of the payroll system due to lack of money to pay the employees. If the company runs out of money (money is a data object), then certain messages must be sent to all concerned objects. The money object must first send a message to the system that it has reached a critical value. The system will then broadcast the message of the termination condition throughout the system. The file which holds the list of employee records will receive the message via a method and either delegate the message, or perform some form of computation upon that message, or both.

3.3 Programming with Procedural Attachments

The idea of procedural attachments comes from recent work in programming languages. The procedural attachment was initially used as a device to patch up program code [7]. If the designer or programmer forgot some procedure or function, they would attach a procedure which carried out that function to the correct location in the program. In the last year XEROX PARC has introduced a structured way to program with access-oriented procedures. Access-oriented procedures are a subclass of procedural attachments.

This work defines a broader scope for procedural attachments which includes access-oriented programming and daemons. Procedural attachments are used by this model to represent procedural knowledge, or that form of knowledge which is hard for the frames to capture and represent. The procedural attachment is used to facilitate three operations in E-ACM: communication between objects, self modification of objects, and as a method for implementing the stimulation and termination conditions as defined for the object in ACM. This section will discuss the definition of procedural attachments, and how they are utilized in the E-ACM system.

The procedural attachment in E-ACM is constructed from two distinct groups: access-oriented attachments and daemons. The procedural attachment defined by this work is a simple algorithm which represents some piece of procedural knowledge integral to the operation of the system. The seductive aspects of procedural attachments are that they run in the background and are unseen by the user, and possess low execution overhead. The difference between the types of procedural attachments is determined by how they will be attached to a particular object or operation. All procedural attachments will have some associated condition which must be triggered (met) before they will fire (execute). There is one exception to this rule - the inactive deamon.

Daemons are the first type of procedural attachment which E-ACM utilizes. The daemons can be divided into two classes: inactive, and active. The inactive daemons used in E-ACM are derived from two areas: frame systems and object-oriented languages (in particular the MIT version of Flavors). Daemons in frames are used for nonmonotonic reasoning. Daemons will calculate information from the lack of information. This use of daemons is a form of default reasoning. Any slot in the frame can have an

attached deamon. Classes also have the ability to inherit daemons from their superclass. This form of inactive deamon defines the first component of the procedural attachment definition in E-ACM.

Inactive and active deamons are derived from the use of daemons as defined in the Flavor's object-oriented programming language. These daemons are attached to methods defined in the object's class. The inactive daemons are called "before" and "after" daemons. Before and after daemons are inactive because they react to any message sent to the attached method, not to a condition. The execution sequence for a method which has attached before and after daemons is the following [16]:

- The before deamon fires first
- The primary method fires
- The after deamon fires
- The value returned is that of the primary method

Each method may have more than one associated before and after deamon. If the method has a list of before daemons, then they are executed in a last-in first-out order. This means that the last attached before daemon will fire first. If the method has a list of after daemons, then they are also executed in a first-in first-out manner [16, 17]. An object can also inherit any before or after daemons defined for a method with the same name, and these daemons are put on the before and after deamon lists.

Active daemons also derive their definition from the Flavors language. These daemons are more powerful because they have conditions attached to them, and they manipulate the message passing facility in Flavors. Flavors

divides these daemons into two types, wrappers and whoppers [17]. Wrappers and Whoppers are associated to methods defined by the object in its class definition. The purpose of the wrapper is to censor incoming messages arguments. The wrapper can test the arguments of a message sent to its associated method, and if the arguments fail to meet the specified conditions, the wrapper stops the message and returns some predefined value. This means the message will not cause the method to fire. This also causes any associated before and after daemons not to execute.

The whopper is similar to the wrapper except that it can alter the message sent to the method. A whopper is a procedural attachment which is associated with a method. The whopper examines the contents of every message sent to its associated method, and if the arguments in the message fail to meet the predefined condition, the whopper can alter the contents of the message or stop the message altogether, depending on the whopper's conditional expression. Both the wrapper and the whopper can also be used to cause the system to take some action before the method fires, even if neither wrapper nor whopper allows the method to fire. The full order of a method's execution with all attached daemons is [17]:

- Wrapper checks message sent to method.
  The wrapper carries out some predefined operation.
  If message meets wrapper condition, then:

  - Fire before daemons
  - Fire primary method
  - Fire after daemons
  - Return value of primary method

  Else ignore message

- Whopper checks message sent to method.
  The whopper carries out some predefined operation.

If message meets whopper condition,
then:
- Fire before daemons
- Fire primary method
- Fire after daemons
- Return value of primary method

Else Alter message contents
Then carry out method

These two powerful daemons are used to facilitate communication between objects, and between objects and the system. Daemons, as defined by Flavors, and frame systems make up the first half of the procedural attachment definition in E-ACM.

Access-oriented procedures complete the definition of procedural attachments in the E-ACM system. The purpose of the access-oriented procedure is to monitor the object's value, and communicate with the stimulation and termination conditions as defined by this system's objects. The researchers at XEROX PARC have proposed a way to structure the use of access procedures in a programming environment [20]. Following is the abstract from their paper describing access-oriented programming [20]:

> Abstract: In access-oriented programming, the fetching or storing of data causes user defined operations to be invoked. Annotated values, a reification of the notion of storage cell, are used to implement active values for procedural activations and properties for structure annotation. The implementation satisfies a number of criteria described for efficiency of operation, and non-interference with respect to other paradigms of programming. The access-oriented programming paradigm has been intergrated with the Loops multi-paradigm knowledge programming system which also provides function-oriented, object-oriented and rule-oriented paradigms for users.

The access-oriented approach attaches procedures to data values, unlike the daemons which attach procedures to methods (operations). Access procedures are triggered by an object, or operation on the object's value, facilitated by either a get and a put on the object's value. An object may have

both a get or a put access procedure attached to its value, and there can be any number of these procedures attached to an object.

These attached operations have conditions which monitor the value of the data object. When the value of the object reaches a certain value the attached procedure will cause a certain action to occur. An example of this type of approach is found in the LOOPS language. In LOOPS there are graphics software devices called gauges. With access procedures these gauges are attached to the values of data objects. The gauge is updated on the screen by the access procedure as the data value is altered [30].

These two types of procedural attachments, daemons and access procedures, are integrated into one procedural attachment definition. This work utilizes the inactive deamon as a nonmonotonic reasoning mechanism. Each slot in the frame which defines an object can have inactive daemons associated with it. The wrapper and whopper daemons are utilized by E-ACM to monitor messages sent to the object, and can alter the state of the message and computation if certain conditions, defined by the wrapper or whopper, are met. Access-oriented procedures are used by E-ACM to monitor the value of an object. If an object's value meets some condition defined in an access procedure, an associated action will be carried out.

The access-oriented procedures will be used to facilitate the stimulation and termination conditions as defined for E-ACM objects (see Chapter Four). When the object's value triggers the access's condition, the attached procedure will notify the stimulation and termination conditions for further action. The procedural attachments as used in this form can be conceptually viewed as interobject communication. The definition for the use of procedural attachments as defined in E-ACM is given in Chapter Four.

3.4 Summary

This chapter discusses and defines the knowledge representation scheme used to define objects in E-ACM. The chapter reviews the knowledge representation scheme of frames, object-oriented programming, and procedural attachments. It also discussed how Frames, the object-oriented concepts of data abstraction, and message passing are integrated to represent the E-ACM system objects. The object-oriented concept of data abstraction treats each object as an encapsulation of information and knowledge. Objects can thus be considered separate entities, and treated in a uniform manner. The Frame scheme is the encapsulation of an object and holds its definition for the data object, including the operations which can be performed upon it. Message passing enforces the concept of data abstraction. The E-ACM model utilizes the message passing concept for interprocess communication. The message passing mechanism is facilitated by the use of code segments defined as methods in the frame, and procedural attachments.

Procedural attachments are defined in E-ACM as a piece of procedural code representing some procedural knowledge. Procedural attachments have a low overhead for execution, and run unseen by the user. Procedural attachments in E-ACM occur in two distinct types: daemons, and access procedures. The procedural attachment facility in E-ACM facilitates inter-object communication, interaction with the system, and object modification.

Chapter Four

4.0 EXTENDED ACM MODEL

This chapter defines the objects, operations and operating system which constitute the E-ACM model. The data objects and operations are defined by the ACM model [1], and are reviewed in Chapter Two of this work. The extensions of the model are defined in depth in this chapter. The concepts for the operating system are derived from recent research done in machine learning [31] and natural language processing [10, 13]. The operating system enforces the data driven concept by using the Petri net model as a model for system control. The Petri net model is enforced by an agenda system, and a blackboard mechanism which is used to control system messages. The integration of the blackboard mechanism and the agenda mechanism constitutes the operating system.

4.1 Fundamental Data Object Definition

This section is devoted to the definition of the data object which inhabits the E-ACM model. The first few sections are dedicated to a review of the generic data object definition given in [1], and defined in Chapter Two of this work. The last three sections are dedicated to the extensions made to the Unger model.

The central definition for the data objects in E-ACM can be found in Unger's model, which takes this definition and extends and modify it. The data object in E-ACM is built around the concept of an abstract data type as defined in the object oriented programming languages [28, 17, 18]. These

objects have lists of authorized operations, the protocol to access system data and knowledge bases, and the protocol necessary to communicate with other objects, i.e., data objects, operations, blackboards, agenda, and the user. The objects also maintain destination lists which hold the paths of execution for particular computations. The objects have the ability, through the use of procedural attachments, to modify themselves. The data abstraction concepts allow the data objects in this model to be treated as a unit, an encapsulation of information and knowledge.

The basic definition of the E-ACM data object is a ten tuple which includes stimulation, designator, attribute, representation, corporality, value, path, history, methods and termination. The extensions to the ACM data object is the path, history, and methods components. The designator, attribute, representation, value, stimulation, and termination components remain intact. The corporality component, specifically the authorization component of corporality, has been extended to include protocols for access to data and knowledge bases.

The designator component is represented physically as a sequence of alphanumeric names which uniquely identifies an object. The designator consists of a four tuple which includes context name, user, instance, and alias. For the E-ACM system only the context name, user, and instance identifiers will be used. Each have a unique meaning related to the physical name. The context of an object's name relates to the environment the object is currently in. The user name is supplied by the user of the system. The instance is used to designate objects which have identical contexts and user names. The designator component is physically represented in E-ACM as a frame:

```
(Designator
  (Context  lisp  user-defined)
  (User     lisp  user-defined)
  (Instance lisp  user-defined) )
```

The user-defined facet connected to the slots of the designator are where default values defined by the systems programmer or the user are placed. Thus if the user fails to give the created object a name, the system will use the predefined (user-defined) value for that slot in the data object. For temporary data objects used by the system, the system will assign the temporary data object designator values.

The next component which defines the E-ACM data object is attribute. A data object's attribute has three components: type, structure, and relationships. Type defines the actual data type the object will be defined as, i.e., integer, boolean, or form. The structure component defines what structures will be used to represent that object. The relationship component is used to define how the current object is related to other objects, i.e., isa, or a kind of (ako) to another object; and define the protocol to communicate with other related objects. The attribute component of the data object is represented in E-ACM by the frame:

```
(Attribute
  (Type              lisp    (daemon to check
                              for valid types))
  (Object-structure  struct )
  (Relationships     lisp  ) )
```

The daemon attached to the type component insures that the type assigned to the data object is a valid type.

The third component of the data object is representation. The representation of a data object consist of a three tuple: location, coding scheme, and packing information. The location of an object corresponds to the

environment in which it was created. The coding scheme represents how the object is physically represented in the system, and is related to the concept in abstract data type definitions which specifies how the object is represented [5]. The coding scheme is defined as one of several types, i.e., list, integer, frame, form, or even file, and must agree with the attribute component type. The packing component stores knowledge about how the data object can be packed for transportation between environments and machines in the network [4]. The representation tuple is represented in E-ACM by the frame :

```
(Representation
 (Location      lisp       )
 (Coding-scheme  set-of-struct (daemon assigns struct
                        depending on assigned
                        type from Attribute
                        component          ))
 (Packing       lisp       ) )
```

When an instance of a data object is created, the system will fill in the Location slot and the Coding-scheme. The packing information is inherited from the type of structure the data object was defined as. For example, an integer is packed differently than a form. The knowledge about how each type is packed is stored in that type.

Corporality is the fourth component of the E-ACM data object definition. This component is defined by the four tuple - longevity, location, replication, and authorization. The longevity component defines how long the value of an object will be bound to that object. The ACM model defines four longevity types:

Static - value fixed at data object creation,
i.e., Data flow models.
Fixed - value fixed at run time,

i.e., Pascal variable.
Dynamic - value may change over time, but a record of past values is kept.
Fluid - value may change over time, but no record of previous values is kept.

A data object can only have one of these longevity types.

The second component of corporality is authorization. ACM defines authorization as the operations which the data object will allow to use its value. The E-ACM model retains that concept, but expands the responsibility of authorization to include protocol information. Protocols define the means for accessing, associated data bases, or knowledge bases. The protocol component holds the information about which data bases, knowledge bases, objects, and system blackboards the current object can access. The protocol component also records how each of the specified objects can be accessed, or communicated with. For example, the data object may have dynamic longevity, meaning a record of past values must be kept. If there are enough values then the data object must have an attached data base. The method for accessing this data base is stored in the protocol component of corporality. The corporality component of the data object is represented in E-ACM by the frame:

```
(Corporality
 (Longevity  lisp  (daemon check for
                    static, fixed,
                    dynamic, fluid))
 (Location    lisp)
 (Replication lisp)
 (Authorization
  (Security  lisp )
  (Protocol
   (Data-base      lisp)
   (Knowledge-base lisp (default protocol))) ) )
```

The knowledge base associated with the object is a system knowledge base which contains, among other things, knowledge regarding the layout of the

office system. This knowledge base will assist the data object if it happens to get lost in the network.

The fifth tuple component defined in ACM is the value component. This component is defined by the ordered pair, information and actual value. The information component is used to store the kind of value this object has such as atomic or aggregation (composite). The atomic types are individual values, i.e., integers, reals, arrays, list. Aggregations are large structure values such as other data objects, combined data objects and operations, forms, and files. An aggregation in this system is analogous to the object oriented concept of a composite object. The information slot is important because atomic structures are handled differently than aggregations.

The actual value component is the actual value stored for the data object if it is an atomic structure. The actual value of an aggregation is the definition for the composite data object. The value component for E-ACM is represented by the frame:

```
(Value
 (Information    lisp (daemon checking for
                 atomic or aggregation
                 structures))
 (Actual-value   lisp ) )
```

The daemon attached to the Information slot is designed to check for valid value types.

The last two components defined in ACM are the stimulation and termination conditions. The stimulation and termination components are similar in that the function of both is to monitor the value of the data object in the actual-value slot of the data object's value component. Stimulation and termination are also similar in that they can cause actions to occur in the

system and can send messages to the system blackboard and other objects. The stimulation and termination components are dissimilar in the kinds of actions which they can cause. The stimulation condition is used by the object to set itself and other objects in motion. This could be to make an object to perform some computation, or for the object for which the stimulation condition was met to perform an action.

When the termination condition is met, it designates the data object's value as no longer valid and not to be used any further in this environment. The a data object, all of its copies, and any children which it spawned must either cease computation or cease to exist, depending on the data object. In the E-ACM model the stimulation and termination conditions are represented by the frame slots :

(Stimulation lisp (procedural attachment))
(Termination lisp (procedural attachment))

The procedural attachment connected to the stimulation or termination condition may be null. The purpose of the procedural attachment is to carry out the necessary operations (i.e., communication) when the condition defined by the stimulation or termination is met.

The E-ACM model defines three new components: path, history, and methods. The path component maintains the execution list for a data object. A data object which has a null path value is an idle, or nonexistent, data object. The object has associated procedural attachments which modify the path list as the object moves through the network of operation nodes. The path component also maintains the path that a data object must follow to traverse the network of work stations in order to complete its

computation.

The path component works in conjunction with the history tuple. The history tuple maintains a recent history of operations carried out on the data object. The history component is also able to record the user who manipulated the object, and if the object's value was altered. When an object visits the first node on its path list, the path list notifies the history component that a computation is about to take place and to take notes. The history records the data object's values prior to execution, and the resulting effects the operation has upon the data object. The path and history components of an E-ACM data object are represented by the frame slots :

(Path lisp (Procedural attachments))
(History lisp (Procedural attachments))

Once the data object has completed its path list for execution, the history component will dump the history of the object into a specified file for inspection by a system coordinator. If an unauthorized access is attempted on the object the history component captures the user's name, and immediately dumps that illegal attempted access into a special error file.

The methods component completes the definition of the E-ACM data object. A data object can have many methods defined for it. The number of methods defined for a particular data object depends upon the types of messages that object is expected to handle. The main purpose of the methods are to handle any message sent to a data object. The object receives messages from other object's by a message sent to one of its defined methods. The collection of the data objects methods defines its protocol and the type of messages it will accept. The methods component of the data

object is represented by the frame slot:

(Methods struct (Attached procedures))

Methods are defined when a new data object definition is added to the system, or may be attached dynamically as a data object's responsibility changes. The slot contains a struct which is necessary for the handling of some messages. The attached procedures are before and after daemon wrapper and whopper - type daemons. The before and after daemons help communication with other objects. The wrappers and whoppers are used to monitor the data object's values and announce changes. All daemons facilitate data object modification and inter-object communication.

The representation for the whole data object definition as represented in E-ACM is the frame:

E-ACM Data Object Definition

```
(Data-object
 (Designator
  (Context          lisp   system-defined)
  (User             lisp   user-defined  )
  (Instance         lisp   system-defined)
  (Alias            lisp   system-defined) )
 (Attribute
  (Type             lisp   (Daemon to check
                            for valid types))
  (Object-structure struct)
  (Relationships    lisp  ) )
 (Representation
  (Location         lisp       )
  (Coding-scheme    set-of-struct (Daemon assigns
                                   struct depending
                                   on assigned type
                                   from  Attribute
                                      component))
  (Packing          lisp       ) )
 (Corporality
  (Longevity        lisp   (Daemon to check
                            for Static,
```

```
                                Fixed, Dynamic, or
                                        Fluid))
    (Location          lisp)
    (Replication       lisp)
    (Authorization
      (Security        lisp)
      (Protocol
        (Data-base     lisp)
        (Knowledge-base lisp  (Default protocol))) ) )
  (Value
    (Information       lisp)  (Daemon to check
                                atomic or aggregation))
    (Actual-value      lisp) )
  (Stimulation         lisp   (Procedural
                                attachment))
  (Termination         lisp   (Procedural
                                attachment))
  (Path                lisp   (Procedural
                                attachment))
  (History             lisp   (Procedural
                                attachment))
  (Methods             lisp   (Procedural
                                attachments)) )
```

Figure 4.1.1

The following is an instantiation of a personnel file data object.

An Example Data Object

```
(Personnel-file
  (Designator
    (Context   VAX 11/780)
    (User      nil )
    (Instance  nil )
    (Alias     nil )
  (Attribute
    (Type              File)
    (Object-structure DBase II)
    (Relationship      Payroll)
  (Representation
    (Location      VAX 11/780)
    (Coding-scheme  ASCII)
    (Packing        nil)
  (Corporality
    (Longevity      Dynamic)
    (Location       VAX 11/780)
    (Replication    0)
    (Authorization
      (Security       (Robin, Sally, George))
      (Data-base       nil)
```

```
   (Knowledge-base  System-KB)
(Value
  (Information  Aggregation)
  (Actual-value Personnel-file-DBII)
(Stimulation   nil)
(Termination   end-of-file)
(Path          (Payroll) )
(History       nil)
(Methods
  (Read    (source code))
  (Write   (source code))
  (Update  (source code))
  (Delete  (source code)) ) ) )
```

Figure 4.1.2

This is a data base which holds all a company's personnel records. The personnel file resides on the VAX 11/780, and at present has no user. It has one object relationship, with the action object Payroll.

This data object has a longevity type of Dynamic, because it will allow users to modify it. The Authorization component of the data object restricts access to Robin, Sally, and George. These names are compared to the User component of the Designator for validity. The personnel data object also has access rights to the system-KB. The protocol for accessing the system-KB is stored in the object as a default value. The information value for the data object is an aggregation. The actual value for the Personnel-file is the data-base itself.

4.2 Action Object Definition

Actions in E-ACM are operations which function on the previously defined data objects. The actions represent nodes in the network which the objects must visit. The definition of the actions are based upon the actions defined in ACM [1]. This section gives a brief review of the action definition as defined in ACM [1] and Chapter Two.

The definition of an E-ACM action is a six tuple - stimulation, material, action, request, termination, and priority. The priority component is the extension to the ACM definition of the action.

The stimulation component is similar to the stimulation component of the data object. The stimulation condition signals an action to start when it should, and works in conjunction with the material component which detects the presence of data objects. The stimulation condition, when met, signals the blackboard, which in turn broadcasts the global state changes to the action and data object. Once the action's stimulation condition is met, the action is placed upon the firing list by the agenda, and its priority is updated. The stimulation is represented in the E-ACM model by the frame slot:

(Stimulation lisp (Communication Daemon))

The communication daemon is responsible for the interaction between the action, the blackboard, the stimulation condition, the priority component, and the material component. The stimulation condition may be null.

The stimulation component must work with the material component. The purpose of the material component is to detect the presence of any data objects the action needs before it can fire. The presence of data objects in the material component triggers the action, if its stimulation condition is met. The number of objects an action requires before it will trigger is fixed, but many data objects may use the operation. This material component is represented in E-ACM by the frame slot:

(Material lisp (Communication Daemon))

The purpose of the communication daemon on the material component is to communicate to the agenda that this action can be placed on the trigger list, and to signal to the stimulation condition that the operation has been triggered.

The action component is the third tuple in the action object definition. The action component is the actual function which performs an operation upon the data objects present in the material list. It is made up of two components - an argument list and the function code. The argument list matches the correct number of data objects found in the material list to the actual function code defined by the action component. It is represented in E-ACM by a special frame called a functional frame. This frame has two slots ; the first holds the argument list for the function, and the second holds the actual arguments needed by the function. The action component is represented by the frame:

```
(Action
  (Operation
      (Argument-list    lisp (Access match))
      (Argument1        lisp)
      (Argument2        lisp)
      " "   " "
      (Argumentn        lisp) )
```

The number of arguments are defined when the function frame is created. Thus when an action object is created, the function frame is attached to the action component slot in the action object frame. The access match is a procedural attachment which, when the operating system tries to retrieve the value from the action slot, fires and matches the material list to the arguments defined by the argument slots. The operation frame then executes the operation and returns a value.

The fourth component in the action definition is the request. The request component defines the list of action which the current action needs to have executed before it can complete its own computation. This list doesn't have to be ordered, because the agenda mechanism handles the ordering of actions. One restriction placed upon the request component is that a child may not request the action of an ancestor. A request will also trigger an action's stimulation condition. The request component is represented by E-ACM as the frame slot:

(Request lisp)

The request component has a default list of null.

The fifth component in the action definition is termination. This component monitors the state of the computation, the blackboard, and its materials component. If the termination condition is met, several actions can be taken by the action object. It could mean that the action is no longer necessary for completion of some computation. If this is the case, the action must signal all actions on its request list to abort. The termination condition may be met by one of the data objects in its material list reaching some critical value, in which case the current action may not be able to use it and must cease computation. The termination component is represented in the E-ACM model by the frame :

(Termination lisp (Communication Daemons))

The communication daemon represents the protocol for the action to broadcast messages to all requested actions, and to communicate the termination of this action to the system blackboard.

The priority component is an extension of the action definition in ACM.

Its purpose is to hold the value of the action's priority. The agenda mechanism uses this component to order and prune the firing and trigger lists. The priority component is represented in this work by the frame slot :

(Priority lisp (deamon modifier))

The procedure attached to the Priority slot is used by the object to keep its priority value current, relative to the rest of the system. For instance, if the action's termination condition is met then the operation updates its Priority slot to allow the operating system to prune it from the list of firing or triggered operations.

The action object definition is represented in the E-ACM model by the following frame:

E-ACM Action Definition

```
(Action
 (Stimulation    lisp  (Communication Daemon))
 (Material       lisp  (Communication Daemon))
 (Action
  (Operation
   (Argument-list   lisp (Daemon match))
   (Argument1       lisp)
   (Argument2       lisp)
   " " " " " "
   (Argumentn       lisp) ) )
 (Request        lisp)
 (Termination    lisp  (Communication Daemons))
 (Priority       lisp  (Daemon modifier) )
```

Figure 4.2.1

An example action is illustrated below. This action is a request by the user for some action :

Payroll: an Example E-ACM Action

```
(Payroll
 (Stimulation   (date = Fri))
```

```
(Material      (Personnel-file Company-Payroll))
(Action        (Read Personnel-file))
(Request       (Check-Print State-tax Fed-tax
                        Social-security))
(Termination   ( (Company-Payroll < 100.0) or
               (Personnel-file = end-of-file))
(Priority      785) )
```

Figure 4.2.2

This example defines an operation which will perform the Payroll function to be carried out. The stimulation condition for the Payroll object is that it must run when the date equals any Friday. The material which must be present are the Personnel-file, and Company Payroll. The action that the Payroll preforms is to read records from the Personnel-file, and send them off for further processing. Payroll requires the actions Check-Print, State-tax, Fed-tax, and Social-security. All these actions carry out some computation, and are defined in the system. The Payroll will terminate if either the Company-payroll runs out of money (money < 100.0), or Personnel-file end-of-file is met. The predefined priority for the Payroll action is 785.

4.3 Operating System Definition

The operation system in E-ACM is based on three concepts: Petri nets, an agenda mechanism, and a blackboard mechanism. The agenda mechanism is integrated with the blackboard system to facilitate the Petri net concept of control. The Petri net concept is utilized in this model to represent the flow of objects through a system of operations and a network of computers. The Petri net model is also chosen because it represents the concept of a data driven model, which is the basis of ACM and E-ACM system control.

Petri nets are utilized on two levels in the E-ACM model. The first level is the network level. Petri nets are used to direct the flow of a data

object through a network of computers. An example will help illustrate this point. Assume a form (which in this model is a data object) in an insurance company must move from the secretary's work station to the agent's work station and back again. The flow of this form through the network can be captured and represented by a Petri net as follows:

A Simple Office Network

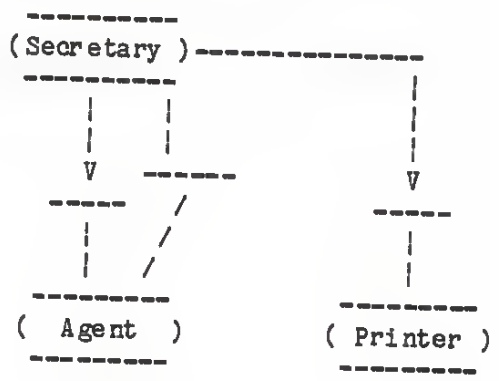

Figure 4.3.1

The nodes in this net represent the stations that the form must visit in a certain order at a certain specified time. All this information is stored internally in the path slot of the data objects definition.

The second conceptual level Petri nets are utilized on is the object level, or the level at which data object interaction with operations and computation occurs. The Petri net at this level illustrates how the objects will interact with operations to perform some computation. To further illustrate this point, consider the previous example of a form at the object level (the form is at the secretary's workstation). This form must have an estimate slot filled in, and a tax calculated from that slot. The secretary must also fill in some information directly on the form, such as company name, date, etc. Here is a Petri net which captures this concept:

An Example Petri net for a Form

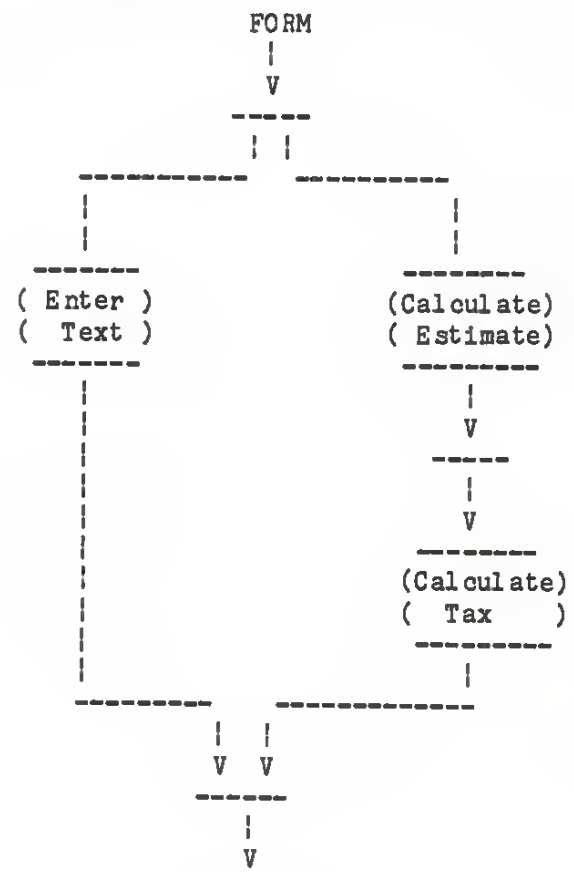

Figure 4.3.2

This net defines the route the form must take while at the secretary's work station. This includes a replication of itself, and a join operation once the operations are completed. The data object stores the object Petri net, with the office network, in its path slot.

The Petri nets are an easy device for a designer to utilize in creating an ordering for a document to follow. The Petri net once designed can then be interpreted by the machine and stored in the respective object's path slot. The agenda and the blackboard concept are integrated to enforce the Petri net design on the system level. The agenda is used to establish an execution order for operations and objects, and uses the blackboard to communicate with the system's objects and the user of the current node (work station) in the network.

The agenda mechanism orders things in a breadth-first manner [10, 13]. The agenda mechanism used in this model is similar to the one used by Lenat to implement his AM and EURISKO systems [31]. In E-ACM it is used for both long and short term scheduling. The agenda mechanism constructs a tree for execution, executing the terminal nodes of the tree first. This execution takes place in a breadth-first manner, depending on the priority mechanism used to order the processes. The function used to calculate an operation's priority in E-ACM depends upon the data objects' availability, the operation's stimulation and termination conditions, the length of the operation from the root operation and a CPU time slice.

The Agenda maintains a firing queue and a trigger queue. The trigger queue is used for long term scheduling. These are operations which objects have requested for use but need the presence of other data objects before they can be fired. The firing queue is used for short term scheduling, and is ordered according to the priority function.

The agenda uses the blackboard mechanism to communicate with the objects in its system and the user using its node in the network. It is partitioned into three regions - system, objects, and user. The blackboard has a controller which only allows access by one user at a time; in other words only one piece of chalk is available. This enforces mutual exclusion of the shared data resource - the blackboard. This use of the blackboard comes from its utilization in the HEARSAY - HEARSAY IV [13, 10], a natural language expert system. The blackboard logically coordinates message traffic in the system. The agenda mechanism, system objects and user communicate to the blackboard via message passing, and the blackboard controller knows where to post the messages.

Conceptually a blackboard mechanism is analogous to a monitor in C-Pascal. The blackboard can handle multiple messages sent to it via a message buffer. The messages that are sent have tags which tell the blackboard who sent which message. The blackboard, like the data object, is an encapsulation of data objects and operations. The blackboard also has methods, which can respond to messages sent to it. The blackboard has associated procedural attachments which help the blackboard maintain its regions and the message buffer. The following is the representation of the blackboard in the E-ACM system:

Frame Representation for the Blackboard

```
(BLACKBOARD
  (Message-buffer  lisp) (Daemon to control buffer
                          access, and size))
  (User-region     lisp) (Daemon to control user
                          buffer))
  (Agenda-region    lisp) (Daemon to control agenda
                          buffer))
  (Object-region   lisp) (Daemon to control objects
                          buffer))
  (Methods         struct) (Daemons)) )
```

Figure 4.3.3

The blackboard controller is defined by the collection of procedural knowledge. Thus, its definition is constructed from the blackboard's protocol and procedural attachments.

The priority list for accessing the blackboard is user, agenda, and then system objects. Highest priority is given to the user, and allows the user to interrupt the system any time to issue a new command or add objects to the system. The agenda uses the blackboard to announce system changes, such as an operation no functioning correctly, by writing messages on the blackboard in the agenda region. Data objects must be made aware of such a change so that they can decide whether or not they need to leave the current system or allow the faulty operation to operate upon them. If the agenda needs to announce an action object failure, it does so by placing the action's name and an attached message, which states the action is not working, upon the blackboard for the system objects to see. The objects use the blackboard to post termination conditions. These conditions are posted by both data objects and operations. The blackboard mechanism broadcasts termination conditions so the agenda may prune terminated operations from its trigger and firing lists and allows data objects to make the corresponding adjustments to the message.

4.4 Summary

The E-ACM system allows for the definition of an intelligent data object model. This chapter has covered the definition for the objects which make up the E-ACM system. The data object and operation receive their definitions from the ACM model. E-ACM extends the ACM data object by adding the components path, history, and methods, and by expanding the responsibility of the authorization component to include the protocol and

access rights to system and network data and knowledge bases. The data object definition is used by the system as a template to create any data object needed in the system. This allows for a consistent system which treats all data objects in a uniform manner.

Operations like the data objects also have a generic definition in this model. The operation definition as defined by ACM was extended by this work to include the priority component. The priority component enables the agenda mechanism to order the operations in a sequence. The operating system is composed of an agenda mechanism and a blackboard mechanism. The agenda and the blackboard mechanism are integrated to enforce the system control concept of a data driven machine based on Petri nets. The blackboard is partitioned into three regions: user, agenda, and objects. The blackboard has a controller which controls access to the three regions, and handles and sorts message traffic to the blackboard. The blackboard and the system objects use a combination of methods and procedural attachments to communicate and modify themselves.

Chapter Five

5.0 Extended ACM Model: An Experimental Environment

This Chapter contains is a description of a partial implementation of the E-ACM model. This implementation is done in the knowledge representation language Pearl, and implemented on the VAX 11/780. Pearl is a frame representation language designed at Berkley [15], and is designed to run under either UCI or Franz Lisp. The concepts from the E-ACM model implemented in this system are the use of frames to represent system objects, the use of a coordinated approach in using procedural attachments, and the system concept of an agenda integrated with a blackboard mechanism. The goal of this implementation is to test the above concepts integrated into a system, and simulate the implementation done by Carter [4]. The problem domain is that of simple arithmetic equations. Following is an account of the implementation definition, including system data objects, operations, and operating system; an example of how the system works; and the problems and questions which the system implementation raises.

5.1 System definition

The partial implementation of E-ACM consists of a data object definition, action definition, and an agenda mechanism integrated with the blackboard mechanism. The agenda integrated with the blackboard mechanism constitutes the operating system for E-ACM. The implementation concentrates on the integration of the agenda and blackboard, and the

use of procedural attachments for inter-object communication and object modification. The problem areas used for testing the implementation are simple Petri net models of mathematical computations. The objects are represented by frames as defined by the Pearl frame system.

The Pearl system is embedded in Franz Lisp. It has one basic structure by which objects are defined and that is the frame. The Pearl frame consists of a frame name and a collection of slots. Each slot must have a name and a type. The allowable types in Pearl are int, Lisp, symbol, struct, and set-of [symbol | struct]. The int type allows only values of type integer to be stored in its slot; its default value is 0. The Lisp type allows any valid s-expression in Lisp to be stored in a slot of this type (i.e., atoms, lists, integers, strings, arrays, etc.).

The symbol type in Pearl is a special structure, and any atom defined as this type must be defined by the function "symbol". A Pearl symbol is analogous to a Pascal constant or literal string, except that the value of a Pearl symbol is the name itself, i.e., (symbol John) defines a symbol whose value is John. Slots of the symbol type have a default value of nilsym.

The slot type of struct allows frames to be embedded in frames, because the generic definition in Pearl for a frame is struct. The default value for a struct is nilstruct.

The last slot type is called set-of. This special type can be used to restrict the slot to a set of valid symbols or structs for that slot. A meta definition for a frame in Pearl is:

Example Pearl Frame

```
(frame-name
   (Int-slot   int)
   (Lisp-slot  Lisp)
```

```
(Symbol-slot symbol)
(Struct-slot struct)
(Set-slot    setof [symbol | struct]) )
```

Figure 5.1.1

Each slot may have many procedural attachments to it. For example, when the user defines a frame, he may specify the default value instead of using the defaults defined by the system. The only restriction placed upon the user in defining defaults is that a slot default must correspond to the slot type. Here is the frame definition with user defined defaults:

Pearl Frame with User Defined Defaults

```
(Frame-name
  (Int-slot     int  25)
  (Lisp-slot    Lisp (a list))
  (Symbol-slot  symbol John)
  (Struct-slot  struct Frame-name-2) )
```

Figure 5.1.2

The defaults are useful to define prototype frames for objects.

To define a meta-frame (description frame), the function "create base" is used. This function binds an atom to a frame defined by the user. The atom thus becomes a frame type, (i.e., like Pascal type definition) with its definition being a structure (frame). The user can then explicitly define default values for the slots. When a base frame is created then any number of instances of that frame may be instantiated, using the "create individual" function. When the user defines an individual frame, that user has the option of specifying which slots he/she will define values for. When an individual frame is created, the slots which receive values override the default values specified for that slot. If the user doesn't specify a value for a given slot, then that slot will use the default value defined for that slot type. For example:

An Example Pearl Class Frame

```
(create base Dog
  (Eye-color  symbol  brown)
  (Legs       int     four )
  (Color      symbol      )
  (Type       Lisp        ) )

(create individual Dog Tucker
  (Legs              3)*
  (Type       cross-bred) )
```

Figure 5.1.3

The first frame defines a frame for a class of objects called dogs. Notice Eye-color, Legs, and Color all have a default value. The second frame defines one individualization of the class of objects called dogs.

bound to the atom Tucker. The Tucker frame has defined the slots Legs as three, and Type as cross-bred. When the Tucker frame is queried for number of Legs it will return the integer value three. If the Tucker frame is queried for its slot value of Eye-color then the default value of green will be returned.

Pearl has a built-in data base system where frames can be stored. This data base uses a hashing scheme to store the frames in buckets. Frames are hashed according to their defined base type. Thus, all frames of type Dog will be stored in the hash bucket associated with the base type Dog. This data base stores individualizations as well as base types.

The Pearl system has a built-in mechanism to handle procedural attachments. These procedural attachments are called "hooks", or "if-added" procedural attachments. Hooks are allowed on both base types, as well as on individualizations. The Pearl procedural attachments, as in LOOPS are

* Question What do you call a Dog with no legs - Answer It doesn't matter he won't come anyway [39].

related to access oriented programming, in that they execute when a frame slot is accessed. The base and individual hooks are listed in [15]. The hooks are triggered when some function tries to operate on the frame. The base hooks can execute either before or after a hooked operation executes. It is up to the system designer to create the conditions which will cause a hook to fire. The designer can stop the operation from executing on that frame if the condition specified by the procedural attachment is not met.

The one restriction placed upon procedural attachments in Pearl is that they must be predefined before they can be used. The following is an example of the use of a base hook and a individual hook.

Pearl Base and Individual Hooks

```
(create base Letter
  (if >individual (putpath * 'Date ate-function)))

  (Date   Lisp)
  (Writer Lisp)
  (Text   struct) )


(create individual Letter Personal
  (Text ^  hook  <put (letter-security * ** =Writer))
```

Figure 5.1.4

This example uses a base hook and an individual hook. The base hook is triggered whenever an individual of the type letter is created. This base hook runs after the individualization is created (">" specifies an after hook). The function of this hook is to put the current date into the individualized Letter just created. The hook used on the individual is placed on a slot; thus, it will only trigger when a certain function tries to access that slot. Individual hooks are associated with slots, whereas base hooks are associated with the entire frame. The hook placed on the Text slot is used only when a put operation is attempted upon that slot. The procedural

attachment associated with Text will execute before the put operation is carried out ("<" specifies before hook).

The purpose of the Text hook is to insure that the user has the correct authorization for using this type of a letter. The "*" means that the attached procedure has access to the current Text slot value. The "**" means that the procedural attachment is dealing with the current frame - Personal. The =Writer allows the hook access to the Writer slot in the Personal frame. The name letter-security is a label for a Lisp function which represents the body of the procedural attachment.

This is a simple introduction into the Pearl frame system. The important components taken from Pearl are the use of defaults, creation of base frames and individual frames, and the intrinsic capability of Pearl to handle base and individual procedural attachments. These components, along with the Lisp environment, were used to create a partial implementation of the E-ACM model. The following section defines the data objects and operations in the Pearl system.

5.2 Object Definitions

The objects defined in E-ACM are only partially used in this implementation. The data objects and operations are represented as Pearl frames. The agenda mechanism is also implemented in Lisp code, and the blackboard is implemented as a Pearl frame. The data object is defined as an eight tuple: designator, attribute, representation, corporality, value, stimulation, termination, and history. The base frame derives its default values from the conceptual graph in Appendix Two. The frame representation of the data object is illustrated in Appendix Four, and is constructed from several frames, creating a composite object. Every component in the E-ACM data

object definition defined by sub-tuples is represented by a base frame. These subcomponents define a template which the user and the system use when creating an individual data object. For instance, the designator slot is defined by the tuple context, user, instance, and alias. It is represented in the implementation by the following frame:

Designator Frame

```
(create base Designator
 (If >individual (putpath * 'Context (systemid)) )
 (If >individual (putpath * 'User ?userid)))
 (If >individual (putpath * 'Instance (ci Instance)))
   (Context   Lisp  )
   (User      Lisp  )
   (Instance  struct)
   (Alias     Lisp  ) )

(create base Instance
   (Spatial int)
   (Time    int)
   (O       int) )
```

Figure 5.2.1

When an individual data object is created, the designator component utilizes three base hooks to fill in its frame slots. The hooks execute after an individual of the designator is made. The first hook puts the system identifier in the Context slot, i.e., KSUVAX1. The second hook stores the user's name in the User slot. When the user signs onto the system, he/she is prompted for an i.d. This i.d. is bound to the system variable $userid. (Pearl denotes variables by attaching a "$" prefix to an atom.) The third hook creates an individualization of the Instance frame, is defined by a three tuple - Spatial, Time, and O. These components have default values of O.

The full data object representation is a frame as illustrated in Appendix Four. The action object, like the data object, is also represented in the system as a Pearl frame. In E-ACM the action object is defined with one additional component - Path. The action is thus defined in the implementation as a seven tuple - Stimulation, Material, Action, Request, Termination, Priority, and Path.

The Path component allows the operation frames to conceptually be considered subprograms. This also shifts the burden of execution from the data object as defined in E-ACM, and back to the operation. The Path component was added to simulate the work done by Carter [4].

The Path slot in an action frame stores the execution path for that operation. The slot itself contains a list of ancestor operations which requested the services of the current action. When the current action executes it uses the Path list as a reference to determine which action will receive those generated values. The Path slot is also used to calculate each action's priority. The action object is represented by the following Pearl frame:

Action Frame Representation

```
(create base Action-Object
  (If >individual (putpath * 'Stimulation (Stim-function)) )
  (If >individual (putpath * 'Action (Kernel-function)) )
  (If >individual (putpath * 'Termination (Term-function)) )

  (Stimulation  Lisp (Blackboard daemon))
  (Material     Lisp (Communication daemon))
  (Action       struct  Kernel)
  (Request      Lisp (Communication daemon))
  (Termination  Lisp (Blackboard daemon))
  (Priority     int )
  (Path         Lisp) )
```

Figure 5.2.2

This frame utilizes three base hooks and two defaults. The base hooks are all associated with the creation of an individual Action object. The individual hooks will all fire after the individualization process has been executed. The first hook (Stim-function) is used to prompt the user for all information concerning what factors will cause that particular action to be triggered. The default stimulation in this implementation is nil, which means that only the presence of the correct number of data objects appearing in its material list is requested. The second hook stores an individualization of an operation from the library of operations. The functions are stored as frames in a system data base. For example, when an add action object is created, the operation-function queries the user for the operation this object is supposed to perform. It then pulls that definition from a data base of action objects and creates an individualization of the frame. The function then stores that frame individualization into the Action slot of the current frame.

The object with which the data object daemons communicate most is the system blackboard. This blackboard coordinates message traffic from Users, the agenda, and system objects. The blackboard daemons specified for both the stimulation and termination components of the data object are

communication daemons. These two daemons communicate to the blackboard the needs of the action. For instance, if its stimulation condition is met then the action must be moved from the trigger list to the firing list. The agenda reads that an action has been triggered and will then move the action. The material daemon informs the stimulation component that the arguments have arrived and that the action may be triggered, thus facilitating a form of inter-object communication. The stimulation daemon then communicates to the blackboard that this action is ready to be placed upon the firing list.

The communication daemon on the Request slot works in coordination with the Agenda to help identify needed actions, which help the actions perform their functions. This daemon issues messages to relevant actions that are present in the system, and communicates to the Agenda (via blackboard) which actions must be pulled from the data base. The procedural attachments attached on the action object are used in a coordinated manner to modify and facilitate action communication with the blackboard, and agenda mechanisms.

The kernel operations are represented by a special frame in the Pearl system, called a functional frame. This frame uses the slots as values for the function which has the same name as the frame type. The base type for Kernel is defined by the following Pearl frame:

```
(create base Kernel
  (Args     Lisp )
  (Operation struct) )
```

This is the base structure the Kernel function manipulates. The args slot in Kernel stores the protocol necessary to match the data objects in the material slot to the arguments needed for the operation frame. The

operation frame is a functional frame pulled from the operation library. It executes the associated function when it is accessed via the kernel frame.

The last object defined in this implementation is the blackboard. The blackboard object is represented by the following Pearl frame:

Blackboard Frame Representation

```
(cb BLACKBOARD
  (User-region   Lisp (Communication Daemon)
                      (Security Daemon)
  (Agenda-region Lisp (Communication Daemon)
                      (Security Daemon)
  (Object-region Lisp (Communication Daemon)
                      (Security Daemon)
  (Methods       Lisp (Attached Daemons)    ) )
```

Figure 5.2.3

This frame consists of four slots. The first three divide the blackboard into three regions. The Methods slot responds to messages sent by the user, agenda, and system objects by directing access to one of the three regions. The User region allows the user to interact with the environment. When the blackboard is accessed the user can view the current state of the environment. The user can then alter the current state by sending messages to the system's objects which will change their stimulation and termination conditions. By accessing the blackboard the user may also inspect an object, or issue new commands to the system.

The agenda slot is utilized by the blackboard to store current system information. This information includes notices about disabled nodes in the office network, failure of an action to complete its function, and the announcement of arrival of new data objects into the current environment. The blackboard only allows the agenda to modify the slot, but allows other objects to read it. This is also true of the user slot; only the user may

modify the User region, but other objects may read from it. The data objects and operations use the Object-region to store information and communicate.

The object region's primary purpose is to store announcements about state changes which directly effect stimulation and termination conditions of system objects. This slot is used, for example, when a data object has reached some critical value which triggers its termination condition. The data object writes the occurrence of the met termination condition onto the blackboard, and all actions with that object in their material lists are pruned from the firing list. The blackboard will announce (by passing messages) that a termination or stimulation condition has been met. This allows for effected data objects and actions to react to the broadcast. However, The blackboard restricts access to this slot to one user at a time.

The method slot is not restricted to one method but to a series of methods. Physically, the methods slot are filled with Lisp code, which is executed when the appropriate method is accessed. Conceptually the methods represent messages the blackboard will respond to and transmit from. Collectively, the messages represent the objects' protocol. The blackboard methods' have two functions in this implementation. The methods primary use is for communication and interaction with system objects (agenda and user included). The second function the methods fulfill is the self modification of the blackboard and its regions. The methods execute in conjunction with procedural attachments.

The procedural attachments are specified in the preceding frame representation of the blackboard. They fulfill three functions. The first function of the blackboard procedural attachments is restriction of access to the three regions. The daemons in charge of security allow only those

objects with proper authorization to modify a particular region. If the object only wishes to read from the region there is no security check. The second function of the procedural attachments is to work with the methods to coordinate the processing of messages sent to the blackboard. The methods and the procedural attachments used in the blackboard collectively define the blackboard controller. This includes sorting the incoming messages, and sending them on to the correct region for processing.

The blackboard is the first component of implementations control system. The second component of the control system is the agenda mechanism. The agenda mechanism conducts a breadth-first search of the action nodes. It gives actions priorities for execution, and then sorts them by decreasing priority values. The agenda then hands up the action with the highest priority to the CPU for execution. The only exception is for actions to be pruned off the firing and trigger lists. The 800 to 850 priority range is reserved for action objects whose termination conditions are met and are no longer needed in the current environment. Any action whose has a priority in that range is pruned from the list by the agenda before all priorities between 100 and 799 which are valid executable actions. The priority function, when applied to the action, constructs an execution tree similar to Carter's implementation.

The nodes of the execution tree are directly related to the priorities given to each action. The priority function is based upon the the presence of data objects, and the distance of the node from the root action. The distance from the root is calculated from the path slot in the action. The path slots are constructed when the user issues a command action. The action is parsed, and stimulation and termination conditions are established. The command action then has its request slot read, and all requested actions are

instantiated, given a priority, and placed on either the trigger or firing list.

This investigation of the request slot is recurrent because it happens for each new action as requested until all requested actions have been placed upon the firing or trigger list. The firing list, if given a form, resembles a tree structure, with the request being the links between nodes and the initial command action being the root. This tree is then executed by the agenda mechanism in a breadth-first manner. The agenda also interacts with the blackboard to store information about the action request and system state. This is useful for actions with long list because the agenda can write the name of the action upon the blackboard and reference it later on when it is needed.

When a command action requests a list of actions most actions are called from the data base, unless they are already present in the system. The command action is analogous to a Pascal program, in that if has a list of requests which are similar to procedure calls in the main program of a Pascal program. The command action has an added dimension: When all procedures have executed, then the program has fulfilled the requirement and is complete. This command action has another dimension which Pascal programs do not have, and that is the stimulation and termination conditions placed on it to specify invocation and termination of the current computation. In a Pascal program, and most other languages, a program must run to completion before it can be normally terminated. In the E-ACM model, including this implementation the termination condition can stop the computation whenever it is met.

The command action posts its termination condition upon the blackboard, and allows the agenda mechanism to prune its firing and trigger lists. The control system in this implementation is constructed from an agenda

mechanism interacting with a blackboard object. The following section is an example of the system working through the execution of a user action command. The listing for this implementation is in Appendix Five.

5.3 Example of system at work

The problem domain, as stated earlier, comes from the field of mathematics. The example problem is the implementation of the quadratic equation. In mathematical form the quadratic equation looks like this:

$$x1 = \frac{-b + SQRT(b^{**}2 - 4 * a * c)}{2 * a}$$

$$x2 = \frac{-b - SQRT(b^{**}2 - 4 * a * c)}{2 * a}$$

The first step in the experimental ACM system is to construct a Petri net to capture the data flow through the operations. The Petri net for the quadratic formula is:

Square Root Petri Net

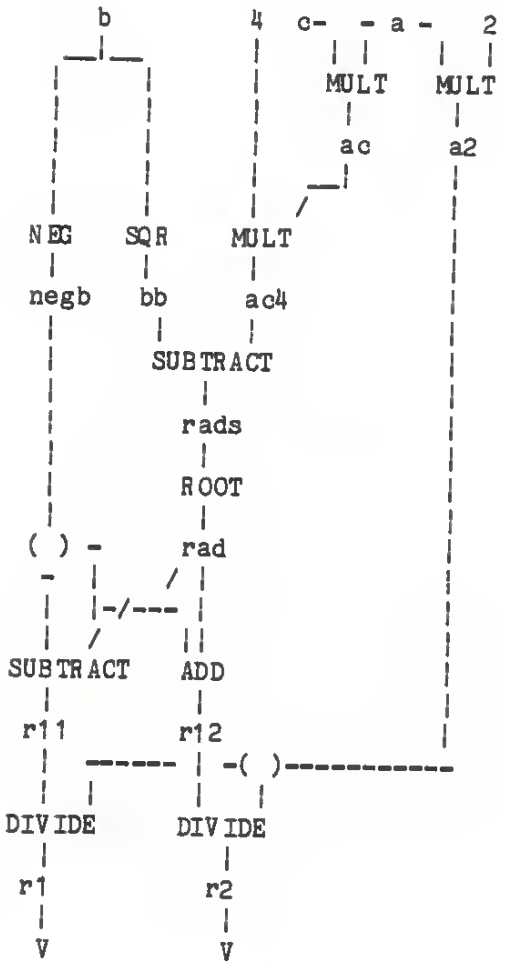

Figure 5.3.1

This Petri net is then translated into the objects needed to execute this net. There are 17 data objects and 12 actions which are utilized to execute this Petri net.

The definitions begin with of the input and output data objects for each action. The first action defined is the NEG operation. This action is represented by the following command:

([] NEG (negb : b) [])

The empty brackets represent null stimulation and termination conditions. The NEG command represents the action frame NEG where its definition is stored.

The parentheses enclose the data objects this function will operate on. The null stimulation condition implies that only the presence of the data objects in its material list is needed to trigger (stimulate) this action. A null termination condition implies that the action will only execute once. The negb data object is the data object which will receive the result of the NEG action. The b data object is the data object which the NEG action will operate on. The updated NEG definition frame after this translation is:

```
    Negative Action Definition

(NEG
  (Stimulation   nil)
  (Material      ((negb) b)
  (Action        NEGATIVE)
  (Request       ( ([] Subtract ( r1 : negb rad) [])
                   ([] Add ( r2 : negb rad) [])     )
  (Termination   nil)
  (Priority      nil)
  (Path          (Quadratic) )  )
```

Figure 5.3.2

The system, when this Petri net is called to execute, assigns the action its priority. The other actions are also defined in this manner (for simplicity, only the NEG action definition is illustrated). These definitions are either created when the Petri net is designed or are already present in the system.

The data objects are also created from the original Petri net if not already present in the system at translation time. For example, if the c data object did not exist then it must be created. The following is a possible definition for the data object c. The system must create an instance of the Data-object frame definition appearing in Appendix Four:

```
Data Object Individualization - C

(ci Data-object c
 (D
  (Designator
    (Context   VAX/780)
    (User      randy)
 (A
   (Attribute
     (Type         real)
     (Obj-struct   Lisp-atom)
     (Relationship (Multiply))
 (R
   (Representation
     (Location    VAX/780)
 (C
   (Corporality
     (1   Fixed)
     (e   (Quadratic))) )
 (V
   (Value
     (Actual-value   20.0)) )
 (Termination   (a * c * 4 >= b*b))
 (History  nil) )
```

Figure 5.3.3

This data object definition defines a data object of type real, representing the value 25.0. The slots which are omitted use the default values defined in the base Data-object definition. The termination condition for this object depends upon the result of 'a' times 'c' times 4 being greater than or equal

to 'b' squared. Thus if this condition is met then the system will not use this data object. Like the action objects the data objects are called from the data base first, then created if needed.

The whole Petri net is given a name, analogous to a program name. This occurs after the net has been defined in terms of the data objects and actions. The form of the command action is:

([] Quadratic(r1 r2 : b c a) [a * c *4 >= b*b])

This is the command action which represents the preceding Petri net. When the user issues this command action, the system will retrieve all necessary actions and objects needed to execute the net. This command action has no stimulation condition; thus, its stimulation is that of being called. The termination condition checks the input to insure that the root will produce real values, instead of imaginary values. This one command will cause the associated Petri net to execute, and return the two roots through r1 and r2. The Petri net can also be initiated by issuing each command individually, specifying the input as well as output data objects. This form is represented as a series of commands:

Square Root Command Sequence

```
([] NEG ( negb : b) [])
([] SQR ( bb : b) [ b < 0])
([] Subtract ( rads : bb ac4) [ ac4 > bb])
([] Multiply (ac : a c) [])
([] Multiply (a2 : a 2) [])
([] Multiply (ac4 : ac 4) [])
([] Divide (r1 : r11 a2) [])
([] Divide (r2 : r12 a2) [])
([] Root (rad : rads) [])
([] Subtract (r12 : negb) [])
([] Add (r11 : rad negb) [])
```

Figure 5.3.4

In this form, however, all actions and data objects must be present in the system. If any of the necessary objects are missing they are dynamically created.

5.4 Conclusions

5.4.1 Problems

The most severe limitation of this implementation is that it focuses almost entirely on the action object of the system. The data objects flow through the operations, but they do not determine their own course of action. Another limitation is that the problem domain is too simple. It would be interesting to extend the implementation to include an office form as it is being processed. Also, the problem of networking several computers together was not addressed by this implementation. Another limitation is that of type checking, since Pearl has a very limited type checker. Pearl can't tell the difference between a char, integer, real, or boolean. The type restrictions must be defined and implemented in the system. This implementation didn't attempt to deal with the issue of garbage collection - in particular that of data objects and operations.

The function in the agenda which calculated the priority is rather simplistic for large applications. The problem faced by a large application is that as the request list grows and the requested operations are put on the firing list, the priority function is biased toward the nodes farthest away from the root. This can lead to an unbalanced tree. This priority system also allows an operation to get exclusive control of the CPU. A better priority would include distance from root and some predefined CPU time slice (depending on the operation).

5.4.2 Extensions

Several extensions need to be made to this implementation to make it user friendly and a complete E-ACM model. The system needs a Petri net editor/translator. An editor/translator removes the burden of having to translate the graph by hand into the frame representations necessary to represent the desired net. The system also needs an editor which would allow modification of the predefined system data objects and operations. This editor could translate a conceptual graph defined by the user into a frame representing the new data object or operation. The editor should also have a graphics interface so that as the actions fire and the objects move through the network, a user can view the execution.

The knowledge base which will contain the needed information about system navigation and access protocol for data bases objects, operations, and other knowledge bases must be designed, and the correct protocol for invocation must be established.

5.4.3. Conclusion

The implementation, though only a partial implementation of the E-ACM model, did bear fruitful results. The frame representation of the system objects worked out effectively, and efficiently. It allowed for easy creation of system objects. The coordination of procedural attachments and methods worked out as planned for object self modification and inter-object communication. The procedural attachments, by being responsible for object modification, removed that responsibility from the operating system.

The operating system was constructed from the integration of an agenda mechanism and a blackboard mechanism. This integration was

easier than anticipated. The implementation also raised many questions which must be answered by further research before a full implementation of E-ACM is attempted. The implementation also illuminated some weak points and components which need to be included in any E-ACM system, i.e., an editor.

Chapter Six

6.0 SUMMARY

This thesis attempts to utilize current concepts in Computer Science, and in particular Artificial Intelligence, to create an office automation tool. A model was constructed by extending the work done by Unger in the ACM model. This thesis uses a Knowledge Engineering approach to extend ACM, resulting in an E-ACM model. This model is a basis for an intelligent data object environment to be used in the office. The intelligent data object system puts the responsibility for system computation upon the data object, and removes it from the system operations and office workers. The following two sections draw conclusions about the thesis, and point out areas of future work for the E-ACM model.

6.1 E-ACM Conclusions

This thesis discusses a method for implementing the termination condition as defined by the ACM model. It discusses a way to integrate the control concepts of an agenda mechanism and a blackboard mechanism to control execution and system communication. The concepts of object oriented programming are used to design and implement the system objects and communication facility (message passing). The object oriented approach creates abstract data types for the E-ACM data objects and actions.

The knowledge representation concept of Frames is used to represent the objects in E-ACM, further enforcing the concept of data abstraction. A Frame creates an encapsulation of information and knowledge which

represents each object. The definition of procedural attachments and a simple programming approach is discussed as a means to facilitate communication and object modification. These concepts integrated together constitute the basis for an office automation system with two characteristics: no master slave relationships exist in the system because the data objects are free to move through the system independently and concurrent operations are accommodated.

6.2 Future Work

This model is not complete. The command interpreter needs to be extended, including the listing of protocols necessary to access the system data and knowledge bases. The model also needs an editor which will allow easy programming of the system. One possible solution is to design an editor with which the user can create Petri nets, and that allows the system to translate a Petri net into system data objects and operations. Another desirable facet of the editor is that it be able to modify system objects and allow for extensions to the system. For example, if the office has a new form to process, the user needs an editor to create that the new form and install it into the system. One possible solution is to have a conceptual language used by both the machine and the user as an interface for creating the objects. The language suggested and utilized in this work is Sowa's conceptual graphs concept. The editor could then take the graph as designed by the user and translate it into a system object. A translator that could be used is the one created by Gary Shultz. The types of attached data base must also be considered. Once the right data base is chosen, then the protocol must be integrated into the E-ACM model.

The paradigm of programming with procedural attachements should be expanded by defining a structured approach to using procedural attachments in a system. The model also needs to be expanded to include the partition of knowledge, and how that knowledge is to be distributed throughout the system. A possible solution is to partition the knowledge in terms of office functions. The division of knowledge by office function would be easier for the users of the system to understand. The loss of object meaning as it is passed from one environment to the next is also a concern. If any component has to be stripped from the data object before it can be sent to the next environment, the object will lose some of its conceptual meaning. This problem is analogous to the problem of translating the intrinsic meaning of natural language into program statements; as the natural language is processed the original meaning becomes less meaningful.

The use of the E-ACM model also lends itself to the concept of program proving. This is facilitated by the use of the stimulation and termination conditions which are placed on data objects and actions. These conditions are similar to the assertions that are placed on segments of code to be proven correct. The author plans to investigate the application of program proof techniques to the E-ACM model.

BIBLIOGRAPHY

[1] Unger E.A., A Natural Model for Concurrent Computation, PhD dissertation 1978 University of Kansas.

[2] Peterson, James L., " Petri Nets ", University of Texas, Austin, Texas. Computing Surveys Vol. 9., No. 3, (Sept 1977), pp 223-235.

[3] Keller, R. M. "Formal Verification of Parallel Programs." CACM, Vol. 19, No. 7 (July 1976), pp. 371-384.

[4] Carter, Cathrine L., "A Concurrency Method on a Network of Corvus Concept Personal Workstations." Master thesis 1984 University of Kansas.

[5] Horowitz, Ellis, "Fundamentals of Programming Languages", (Computer Science Press 1984), Second Edition, pp 233-262, 287-320, 373-393, 395-413.

[6] Pressman, Roger S., "Software Engineering A Practitioner's Approach", McGraw-Hill Book Co., Copyright 1982, pps 178-204.

[7] Hartley, Roger T., "Topics in Artificial Intelligence", CS890 KSU, Summer 1985.

[8] Hendrix, G. G., "Encoding Knowledge in Partitioned Networks", Associative Networks: The Representation and use of Knowledge in Computers, Findler, NY(ed), Academic Press, New York, NY, 1979.

[9] Winston, Patrick H., "Artificial Intelligence", Addison-Wesley Publishing Co., 2nd edition, 1984.

[10] Barr, Avon, Feigenbaum, Edward A., "Handbook of Artificial Intelligence",(William Kaufman, Inc 1982) vol 1.

[11] Barr, Avon, Feigenbaum, Edward A., "Handbook of Artificial Intelligence",(William Kaufman, Inc 1982) vol 2.

[12] Cohen, Paul R., Feigenbaum, Edward A., "Handbook of Artificial Intelligence",(William Kaufman, Inc 1982) vol 3.

[13] Rich, Elaine, "Artificial Intelligence",(McGraw-Hill Co. 1983) pp 173-292.

[14] Sowa, John F., "Conceptual Structures Information Processing in Mind and Machine", (Addision Wesley 1984), pp 69- 137.

[15] Deering, Michael, Faletti,Joseph., Wilensky,Robert., "Using the PEARL AI Package (Package for Efficient Access to Representations in Lisp). Computer Science Division University of California, Berkeley. February 1982.

[16] Allen, Ellizabeth M., Trigg, Randall H., Wood, Richard J., "The Maryland Artificial Intelligence Group Franz Lisp Environment", Maryland Artificial Intelligence Group Department of Computer Science College Park MD, 1983.

[17] Symbolics manual for Symbolics machine. "Flav Objects, Message Passing, and Flavors"., Massachusetts Institute of Technology, Feb 1984, version 5.0.

[18] Bobrow, D., & Stefik, Mark, The LOOPS manual. Xerox PARC, December 1983.

[19] Reiter, R. "On reasoning by defaults", TINLAP-2, pp 210-218.

[20] Bobrow, Daniel G., Kahn, Kenneth M., Stefik, Mark J., "Integrating Access-Oriented Programming in to Multiparadigm Environment". 30 sept 85 XEROX Palo Alto, Ca. 94304.

[21] Lochovsky F.H. Tsichritzis D.C., "Office Information Systems: Challenge for the 80's", IEEE vol 88, no.9 1980 pp 1054-1055.

[22] Zisman, Michael D. "Office Automation : Revolution or Evolution?", Sloan Management Review Spring 1978 pp 1-16.

[23] Tanenbaum, Andrew S., "Network Protocols", ACM 1981, Computing Surveys, vol 13, no. 4, December 1981 pg 453-489.

[24] Kleinrock, Leonard, "Distributed Systems", Communications of the ACM, November 1985 vol 28, no 11 pp 1200 - 1213.

[25] Ellis, Clarence A. Nutt, Gary J., "Computer Science and Office Information System", June 1979, XEROX PALO ALTO Research Center.

[26] Martin P., Tsichritzis, D., "A Message Management Model", Computer Systems Research Group, pp 63 - 77.

[27] Gibbs, Simon, Hogg, John, Nierstrasz, Oscar, Rabitti, Fausto, Tsichritzis Dennis., "A System for Managing Structured Messages", IEEE Transactions on Communications, vol 30 no 1 Jan. 1982.

[28] "The Smalltalk-80 System", The Xerox Learning Research Group, BYTE Aug. 1980 volume 6 number 8, pps 36-48.

[29] Baker, Henry Jr., Hewitt, Carl, "Actors and Continuous Functionals", MIT/LCS/TR -194, MASSACHUSETTS INSTITUTE OF TECHNOLOGY LABORATORY FOR COMPUTER SCIENCE, Dec 1977.

[30] Stefik, M., Bobrow, Daniel G., "Object Oriented Programming Themes and Variations", A.I. Magazine, vol 6. no 4. Winter 1986.

[31] Lenat, Douglas B., "Learning by Discovery", Machine learning an Artificial Intelligence Approach.(Tioga Publishing Co. 1983) pp 243 -307.

[32] Hansen, Per Brinch, The Architecture of Concurrent Programs. (Prentice-Hall Inc. 1977) pp 52.

[33] Charniak, Eugene, Riesbeck, Christopher K., McDermott, Drew V., "A.I. Programming", (New York, Halstead Press division, Wiley 1980).

[34] Steele, Guy L., Common Lisp The Language. (Digital Equipment Corporation 1984).

[35] Hayes-Roth, Frederick, Lenat, Douglas B., Waterman, Donald A., "Building Expert Systems" (Reading, Mass: Addison-Wesley Pub 1983)

[36] Dahl, O., Nygaar, K., "SIMULA-An ALGOL- Based Simulation Languages". Comm. ACM, 9, 9, 1966 pp 671-678.

[37] The Bear, Fozzy, "Wacka Wacka - a bear and his jokes". (Jim Henson Pub. INC. NY, NY 345084) pp 1-3.

APPENDIX 1 ACM Definition

prototype for Data-Object: x is

```
[Data-Object: *x] -
  (Char)->[Designator] -
          (Attr)->[Context: nil]
          (Attr)->[User: nil]
          (Attr)->[Instance:*y]-
                  (Cont)->[Spatial: 0]
                  (Cont)->[Time: 0]
                  (Cont)->[O : 0],
          (Attr)->[Alias: nil],
  (Char)->[Attribute]-
          (Attr)->[Type: {Int |
                          Char |
                          Real |
                          Boolean |
                          Aggregation}]
          (Attr)->[Object-structure]-
                  (Cont)->[Structures: {*}],
          (Attr)->[Relationship: nil],
  (Char)->[Representation]-
          (Attr)->[Location: nil]
          (Attr)->[Coding-scheme: nil]
          (Attr)->[Packing: nil-struct],
  (Char)->[Coporality]-
          (Attr)->[Logevity: {Statid
                              Fixedl
                              Dynamid
                              Fluid}]
          (Attr)->[Location: nil]
          (Attr)->[Replications: 0]
          (Attr)->[Authorization]-
                  (Cont)->[Security: lisp]
  (Char)->[Value]-
          (Attr)->[Information: {*}]
  (Char)->[Stimulation: nil]
  (Char)->[Termination: nil]
```

prototype for Action-Object :*y is

```
[Action: *y]-

   (Char)->[Stimulation: nil]
   (Char)->[Material: nil]
   (Char)->[Action: struct]
   (Char)->[Request: nil]
   (Char)->[Termination: nil]
   (Char)->[Priority: nil].
```

APPENDIX 2 E-ACM Conceptual Graphs

prototype for Data-Object: x is

```
[Data-Object: *x] -
  (Char)->[Designator] -
          (Attr)->[Context: nil]
          (Attr)->[User: nil]
          (Attr)->[Instance:*y]-
                   (Cont)->[Spatial: 0]
                   (Cont)->[Time: 0]
                   (Cont)->[O : 0],
          (Attr)->[Alias: nil],
  (Char)->[Attribute]-
          (Attr)->[Type:{Int|
                         Char|
                         Real|
                         Boolean|
                         Aggregation}]
          (Attr)->[Object-structure]-
                   (Cont)->[Structures: {*}],
          (Attr)->[Relationship: nil],
  (Char)->[Representation]-
          (Attr)->[Location: nil]
          (Attr)->[Coding-scheme: nil]
          (Attr)->[Packing: nil-struct],
  (Char)->[Coporality]-
          (Attr)->[Logevity: {Statid
                              Fixed|
                              Dynamid
                              Fluid}]
          (Attr)->[Location: nil]
          (Attr)->[Replications: 0]
          (Attr)->[Authorization]-
                  (Cont)->[Security: lisp]
                  (Cont)->[Potocol]-
                           (Cont)->[Data-base: *d]
                           (Cont)->[Knowledge-base: *k],
  (Char)->[Value]-
          (Attr)->[Information: {*}]
          (Attr)->[Actual-value: struct],
  (Char)->[Stimulation: nil]
  (Char)->[Termination: nil]
  (Char)->[Path: nil]
  (Char)->[History]-
          (Attr)->[Operations: nil]
          (Attr)->[Past-values: nil]
          (Attr)->[PVDB: *do]
  (Char)->[Methods: {*}].
```

prototype for Action-Object :*y is

```
[Action: *y]-

   (Char)->[Stimulation: nil]
   (Char)->[Material: nil]
   (Char)->[Action: struct]-
            (Cont)->[Kernal: struct]-
                     (Cont)->[Arg-Protocol]
                     (Cont)->[Operation: {*}],
   (Char)->[Request: nil]
   (Char)->[Termination: nil]
   (Char)->[Priority: nil].
```

prototype for BLACK-BOARD :*z is

```
[BLACK-BOARD: *z]-
   (Char)->[User-Region: nil]
   (Char)->[Agenda-Region: nil]
   (Char)->[Object: nil]
   (Char)->[Methods: {*}].
```

APPENDIX 3 E-ACM Frame Representation

```
;Object Definition ———————————————— ;This is the frame representation for the ; E-ACM model data object

;Designator tuple definition————————————

(cb Instance
   (Spatial int)
   (Time    int)
   (O       int) )


(cb Designator
   (If > individual (putpath * 'Context (systemid)) )
   (If > individual (putpath * 'User ?userid)) )
   (If > individual (putpath * 'Instance (ci Instance)) )
   (Context  lisp)
   (User     lisp)
   (Instance struct)
   (Alias    lisp) )

;end Designator————————————————————

;Attribute tuple definition————————————

(cb Attribute
   (Type         symbol)
   (Obj-struct   setof struct)
   (Relationship lisp) )

;end Attribute————————————————————

;Representation tuple definition————————

(cb Representation
   (Location      lisp )
   (Coding-scheme lisp)
   (Packing       struct) )


;end Representation————————————————————

;Coporality tuple definition————————————

(cb Longevity
   (Static  lisp)
   (Fixed   lisp)
   (Dynamic lisp)
   (Fluid   lisp) )
```

```
(cb Authorization
   (Security  lisp)
   (Protocol  struct) )

(cb Protocol
   (Data-base       setof struct)
   (Knowledge-base  setof struct) )

(cb Corporality
   (l    struct Longevity)
   (loc  lisp)
   (r    int 0)
   (e    struct Authorization) )
 .sp 2 ;end Corporality----------

;Value tuple definition----------

(cb Value
   (Information  setof struct)
   (Actual-value struct) )


;end Value----------

;History Definiton----------

(cb History
  (Operations  lisp   )
  (Past-values lisp   )
  (PVDB        struct ) )

;end History----------


;Data-Object Definition----------

(cb Data-Object
   (if >individual (putpath * 'D (ci Designator)     )
      >individual (putpath * 'A (ci Attribute)      )
      >individual (putpath * 'R (ci Representation) )
      >individual (putpath * 'C (ci Corporality)    )
      >individual (putpath * 'V (ci Value))         )
   (D          struct )
   (A          struct )
   (R          struct )
   (C          struct )
   (V          struct )
   (Stimulation lisp )
   (Termination lisp )
   (Path        lisp )
   (History     lisp )
   (Methods     setof struct) )
```

```
;end Data-Object Definition----------

;Action definitions ----------

(cb Kernal
   (Args      lisp)
   (Operation struct) )


(cb Action-Object
  (If >individual (putpath * 'Stimulation (Stim-function))
      >individual (putpath * 'Action (Kernal-function)   )
      >individual (putpath * 'Termination (Term-function)))

   (Stimulation lisp)
   (Material    lisp)
   (Action      struct Kernal)
   (Request     lisp)
   (Termination lisp)
   (Priority    int) )

;end Action----------

;Blackboard definition----------

(cb Blackboard
   (User-region   lisp)
   (Agenda-region lisp)
   (Object-region lisp)
   (Methods       setof struct) )
```

APPENDIX 4 Implementation Object Representation

```
;Object Definition ---------- ;This is the definition for the
ACM model ;data object

;Designator tuple definition----------

(cb Instance
   (Spatial int)
   (Time    int)
   (O       int) )


(cb Designator
   (Context  lisp)
   (User     lisp)
   (Instance struct)
   (Alias    lisp) )

;end Designator----------

;Attribute tuple definition----------

(cb Attribute
   (Type         symbol)
   (Obj-struct   setof struct)
   (Relationship lisp) )

;end Attribute----------

;Representation tuple definition----------

(cb Representation
   (Location      lisp )
   (Coding-scheme lisp)
   (Packing       struct) )


;end Representation----------

;Coporality tuple definition----------

(cb Longevity
   (Static  lisp)
   (Fixed   lisp)
   (Dynamic lisp)
   (Fluid   lisp) )


(cb Authorization
   (Security  lisp) )
```

```
(cb Corporality
   (l  struct Longevity)
   (loc lisp)
   (r  int  0)
   (e  struct Authorization) )

;end Corporality----------

;Value tuple definition----------

(cb Value
   (Information  setof struct)
   (Actual-value lisp) )

;end Value----------

;History Definiton----------

(cb History
  (Operation  lisp   )
  (Past-value lisp   )
  (PVDB       struct ) )

;end History----------

;Data-Object Definition----------

(cb Object
  (if >individual (putpath * 'D (ci Designator) )
      >individual (putpath * 'A (ci Attribute) )
      >individual (putpath * 'R (ci Representation) )
      >individual (putpath * 'C (ci Corporality) )
      >individual (putpath * 'V (ci Value)) )

   (D          struct )
   (A          struct )
   (R          struct )
   (C          struct )
   (V          struct )
   (Stimulation lisp )
   (Termination lisp )
   (History     lisp ) )

;end Data-Object Definition----------
```

```
;Action definitions ————————————————

(cb Kernal
   (Args lisp (Arg1 Arg2))
   (Operation struct) )


(cb Action
  (If >individual (putpath * 'Stimulation (Stim-function))
     >individual (putpath  * 'Action (Kernal-function))
     >individual (putpath * 'Termination (Term-function)))

   (Stimulation lisp)
   (Material    lisp)
   (Action      struct Kernal)
   (Request     lisp)
   (Termination lisp)
   (Path        lisp)
   (Priority    int) )


;end Action ————————————————————————


;Black-board Definition ————————————————

(cb BlACK-BOARD
  (User-region   lisp)
  (Agenda-region lisp)
  (Object-list   lisp)
  (Methods       lisp) )

;end Black-board definition ————————————
```

APPENDIX 5 Implementation Control System

```
;This function creates and updates actions

(defun execute-action (action)
  (putpath (eval (caar (getpath (eval action) 'Material)))
           '(V Actual-value)
           (fire-action action))
  (putpath (eval (caar (getpath (eval action) 'Material)))
           'History
           action) )


;This function evaluates (fires) the action

(defun fire-action (action)
   (evalfcn (getpath (eval action) '(Action Operation))) )


;This function creates a new object dynamically during program execution.

(lambda (action-name)
      (set (implode (getpath action-name 'Material))
           (ci Object)) )


;This is a predicate which test for priority ranges between 800 and 900

(defun priority-prun (action)
  (cond ((and (< 799 (getpath (eval action)
                              'Priority))
              (> 900 (getpath (eval action)
                              'Priority))) t) ) )

;Agenda which drives the execution of a program

(defun Agenda-aux (pgm-list)
     (cond ((null pgm-list) nil)
           ((priority-prun (car pgm-list))
              (Agenda-aux (permute-action *pgm-list*)))
           ((materialp-in (car pgm-list) *obj-list*)
              (car pgm-list))
           ((materialp-out (car pgm-list) *obj-list*)
                 (getpath (eval (car pgm-list)) 'Action)
                 nil)
           (t (Agenda-aux (reprioritize pgm-list))) ) )
```

```
(defun Agenda (pgm-list)
    (setq *pgm-list* pgm-list)
    (prog ()
      loop  (let ((action (Agenda-aux *pgm-list*)))
                (black-board action 's)
                (cond ((eq action nil) (return 'done))
                      (t (retrieve-value action)
                         (execute-action action)
                         (permute-alist *pgm-list*)
                         (permute action)) )
                (black-board action 't) )
                (go loop) ) )


;This function retrieve the values for the action to fire


(defun retrieve-value (action)
   (retrieve-vaux action
                (getpath (eval action)
                         '(Action Args))
                (cdr (getpath (eval action)
                              'Material))) )


(defun retrieve-vaux (action arg-list obj-list)
  (cond ((null arg-list) t)
        (t (putpath (eval action)
                    '(Action Operation ,(car arg-list))
                    (cond ((numberp (car obj-list))
                           (car obj-list))
                          (t (getpath (eval (car obj-list))
                                      '(V Actual-value))))  )
           (retrieve-vaux action (cdr arg-list)
                          (cdr obj-list)) ) ) )



;This is the initial black-board definition

(defun black-board (action type)
   (cond ((eq type 's) (black-board-s action))
         ((eq type 't) (black-board-t action))) )
```

```
(defun black-board-s (action)
  (apply (cadr (getpath (eval action) 'Stimulation))
         (list (getpath (eval (car (getpath (eval action)
                                            'Stimulation)))
                        '(V Actual-value))
               (getpath (eval (caddr (getpath (eval action)
                                              'Stimulation)))
                        '(V Actual-value)) )
         ) )


(defun black-board-t (action)
   (cond ((cond-test action)
          (send-term (cons action
                           (append   (getpath (eval action)
                                              'Path)
                                     (getpath (eval action)
                                              'Request))) ))
         (t nil)) )


(defun cond-test (action)
  (apply (cadr (getpath (eval action) 'Termination))
         (list (getpath (eval
                         (car (getpath (eval action)
                                       'Termination)))
                        '(V Actual-value))
               (getpath (eval
                         (caddr (getpath (eval action)
                                         'Termination)))
                        '(V Actual-value)) ) ) )


(defun send-term (action-list)
  (cond ((null action-list) nil)
        (t (putpath (eval (car action-list))
                    'Priority
                    850)
           (send-term (append (cdr action-list)
                              (getpath (eval
                                        (car action-list))
                                       'Request)) ) ) ) )
```

```
; This an example of how to use the function frames

(de Add-one (a1 a2)
  (+ a1 a2))

(cf Add-one
  (Arg1 lisp)
  (Arg2 lisp))

(ci Add-one plus1
  (Arg1 34)
  (Arg2 10))

(ci Add-one plus2
  (Arg1 19)
  (Arg2 100))

(de Mult (a1 a2)
   (* a1 a2) )

(de Subt (a1 a2)
    (- a1 a2) )

(de Div (a1 a2)
    (/ a1 a2) )

(cf Mult
 (Arg1 lisp)
 (Arg2 lisp) )

(cf Subt
 (Arg1 lisp)
 (Arg2 lisp) )

(cf Div
 (Arg1 lisp)
 (Arg2 lisp) )


;This hook prints out the the final output for a program

(defun print-output (m-list)
      (msg "The values from this program are")
      (terpr)
      (print-outaux (car m-list))
      '(*done* (program completed)) )
```

```
(defun print-outaux (m-list)
  (cond ((null m-list) t)
        (t (msg "   " (car m-list) " = "
                (getpath (eval (car m-list))
                         '(V Actual-value)) )
           (terpr)
           (print-outaux (cdr m-list)))) )

;This hook update an objects History slot

(defun history-update (object h-list)
   (putpath object 'History (cons action h-list)) )
```

```
(defun op-system ()
  (msg "Welcome to the E-ACM system") (terpr)
  (msg "Date created 1 Feb 1986") (terpr)
  (exec date) (terpr)

  (prog ()
    (msg "—> ")
    loop (let ((command (read)))
          (cond ((quitp command)
                 (return (final-message)))
                ((equal command 'diagnostic)
                 (diagnostic)
                 '(we now return you to the system))
                (t (parse command)
                   (system (getpath (eval
                                      (cadr command))
                                    'Request)
                           (cdr  (getpath (eval
                                            (cadr command))
                                          'Material)))
                   (getpath (eval (cadr command))
                            'Action))
                )
          )
         (terpr)
         (msg "—> ")
         (go loop) ) )


(defun quitp (command)
  (caseq command
    (q t)
    (quit t)
    (d t)
    (done t)
    (exit t) ) )


(defun system (act-list obj-list)
      (global *obj-list*)
      (setq *obj-list* obj-list)
      (prioritize act-list)
      (let ((action-list (sort act-list 'comp-fnc->)))
          (Agenda action-list)) )
```

```
;This function intializes all action on the act-list

(defun prioritize (pgm-list)
      (mapcar 'priority-fill pgm-list))


;This function calculates an actions priority


(defun priority-fill (action)
  (putpath (eval action) 'Priority
          (* (+ (* (length (getpath (eval action)
                                    'Path))
                   10)
                (if (materialp-in action *obj-list*)
                  then (+ (length (getpath (eval action)
                                           'Material))
                          10)
                  else (length (getpath (eval action)
                                        'Material))))
             10))
   action )

; This is the compare function used to prioritize the agenda's active list

(defun comp-fnc-> (frame1 frame2)
  (> (getpath (eval frame1) 'Priority)
     (getpath (eval frame2) 'Priority)) )


;This function is designed to search for the presence of data in a list

(defun materialp-in (frame data-list)
  (let ((needed-mat (cdr (getpath (eval frame)
                                  'Material))) )
     (cond ((null needed-mat) nil)
           (t (materialp-aux needed-mat data-list))) ) )


(defun materialp-out (frame data-list) t)
  (let ((needed-mat (car (getpath (eval frame)
                                  'Material))) )
      (cond ((null needed-mat) nil)
            (t (materialp-aux  needed-mat
                               data-list))) ) )
```

```
(defun materialp-aux (needed-mat data-list)
    (cond ((null needed-mat) t)
          ((numberp (car needed-mat))
                (materialp-aux (cdr needed-mat)
                               data-list))
          ((memq (car needed-mat) data-list)
                (materialp-aux (cdr needed-mat)
                               data-list)) ]


;This is the function which reads in a command from the key- board and
parses it

(defun test-r ()
  (let ((command (read)))
      (parse command)) )

(defun parse (command)
  (let ((action (cadr command)))
      (cond ((null command) nil)
            (t (stim-parse (car command) action)
               (term-parse (cadddr command) action)
               (action-parse (caddr command) action)) )
       action )  )

(defun stim-parse (stim action)
  (putpath (eval action) 'Stimulation  stim) )


(defun term-parse (term action)
  (putpath (eval action) 'Termination term) )


(defun action-parse (ob-list action)
  (putpath (eval action)
           'Material
           (cons (output-value-parse ob-list)
                 (input-value-parse ob-list))) )


(defun output-value-parse (l)
  (cond ((null l) nil)
        ((eq (car l) ':) nil)
        (t (cons (car l)
                 (output-value-parse (cdr l))) ) ) )


(defun input-value-parse (l)
  (cond ((null l) nil)
        ((eq (car l) ':) (cdr l))
        (t (input-value-parse (cdr l))) ) )
```

```
(defun final-message ()
     (msg "Have a good day") (terpr)
     (exec date) (terpr))


(defun permute (action)
   (setq *obj-list* (append *obj-list*
                    (car (getpath (eval action)
                                  'Material)) ) ) )


(defun diagnostic ()
  (prog ()
    loop  (msg " Select one of the following ")
          (terpr)
          (terpr)
          (msg "1) Show-object  2) Show-action 3) exit")
          (terpr)
          (msg " Enter choice  :  ")
          (let ((choice (read)))
              (caseq choice
                (1 (show-object))
                (2 (show-action))
                (3 (return nil)) ) )
          (go loop) ) )
```

A KNOWLEDGE ENGINEERING APPROACH TO ACM

by

RANDY G. HAHN

B.S., Kansas State University, 1984

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

Knowledge representation is currently a central research issue in Computer Science. Knowledge properly represented and integrated into a computer system can enhance its processing capability. In fact, the ideal computer system may be one in which the user issues commands to the system, and the system carries out those commands with out any further assistance. Such a system could also handle many tasks at once and communicate with other computers to accomplish other tasks.

In this work a new model for representation of knowledge through data objects which have inherent intelligence is developed. The data objects are extended to utilize processing instructions, routing information among computing agents, with the option of protocols to access system wide knowledge bases. The knowledge bases are based upon the Artificial Intelligence techniques developed from Knowledge Engineering, and semantic graphs. Objects will communicate via message passing or through a common blackboard. A prototype form of this model has been implemented in the PEARL frame system on the VAX 11/780. The model and prototype implementation begins to assume the feasibility of an autonomous data object approach to distributed system design.