AN ABSTRACT OF THE THESIS OF

Sankar L. Chakrabarti for the degree of Master of Science in Computer Science presented on April 12, 1985.

Title : STATES, SIDE EFFECTS AND MULTIPLE CONTEXTS IN PROGRAMMING LANGUAGES

Abstract approved: Redacted for Privacy

M.J. Freiling

In von Neumann Languages, side effects occur if one or more non local variables change value(s) during the execution of a procedure or a function. Side effects can occur only if the programming language provides a notion of memory (or state) where the effect will be stored. Side effects complicate the semantics of such languages. As a result programs are often hard to prove or verify in languages permitting side effects. The applicative languages eliminate the problem of side effects altogether by removing the concept of memory or state. Consequently,these languages have simpler semantics and applicative programs are easier to to prove. In this thesis we explore the design of the programming language EML (for Environment Modeling Language). EML retains the the notion of memory and restricts but does not eliminate side effects. Restriction of side effects is enforced by eliminating the notion of global accessibility to a data item and by the introducing the concept of a 'consumable' variable. In this model a variable once accessed by a procedure is assumed to be consumed in the calling environment. A consumed variable may not be used by any other procedure till its value is regenerated by the execution of some other procedure. Instead of global accessibility, in EML data exchange is limited only between neighboring environments, i.e an EML procedure can accept and return data only to the procedure which invokes it. We show that these restrictions simplify the semantics of assignment based languages and improve the provability of EML programs. An approach to verifying EML programs employing the notion of equivalent function is discussed. EML should be useful for programming and verifying properties of programs intended to model and manipulate environments.

STATES, SIDE EFFECTS AND MULTIPLE CONTEXTS IN PROGRAMMING LANGUAGES

by

Sankar L. Chakrabarti

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master Of Science

Completed April 12, 1985

Commencement June 9, 1985

APPROVED:

Redacted for Privacy

Assistant Professor (Computer Science), in charge of major

Redacted for Privacy

Chairman , department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented: April 12, 1985

TABLE OF CONTENTS

CHAPTER 1
STATES, SIDE EFFECTS AND MULTIPLE CONTEXTS
IN PROGRAMMING LANGUAGES
INTRODUCTION 1

CHAPTER 2
CONSUMABLE VARIABLES & UNIDIRECTIONAL CHANNELS:
MODEL OF A VIRTUAL MACHINE 5
2.1 How Side Effects Occur In
In von_Neumann Languages 5
2.2 How Side Effects Could Be Controlled
In von_Neumann Languages 9
2.3 Consumable Variables 12
2.4 Notion Of A Two State Memory System 14
2.5 Unidirectional Channels Linking
Computation And Memory 17
2.6 EML_machine vs. Other Virtual Machines 19

CHAPTER 3
SYNTACTIC SPECIFICATION OF EML 21
3.1 EML_entities 21
3.2 EML_Programs/User_defined Operators 29
3.3 Rules Of Usage Of Various Entities 33
3.3.1 Use Of 'receive' And 'send' operators 34
3.3.2 Use Of Operators 36
3.3.3 Use Of Variables 37
3.3.4 Use Of Constants 37
3.4 Loop Statement 38

CHAPTER 4
SEMANTIC SPECIFICATION OF EML 47
4.1 Overview 47
4.2 Definitions Of Some Useful Terms 47
4.3 Working Of The EML Machine: Informal Description 48
4.3.1 Sequential Execution Of Statements 48
4.3.2 Executability Of A Statement 49
4.3.3 Effect Of Execution Of A Statement 51
4.3.4 Values Generated By Execution Of A Statement 55
4.4 Useful Functions 55
4.5 Semantic Specification Of The EML Language 69
4.5.1 Formal Definitions 70
4.5.2 Abstract Syntax 71
4.5.3 Effect Of Executing An EML Operator 73
4.6 Meaning Of Executing An Operator 74

CHAPTER 5
VERIFICATION OF EML PROGRAMS 86
5.1 From Program To Expressions 86
5.2 Verification Of An EML Operator 88
5.2.1 Example 1 89
5.2.2 Example 2 95
5.2.3 Example 3 108
5.2.4 Example 4 110

CHAPTER 6
DISCUSSION 113
6.1 Program Verification In General: EML vs. Imperative Languages 113
6.2 Dual View Of Operators 120
6.3 Efficiency Of Data Tarnsfer Between Contexts 122
6.4 Position Of EML In The Family Of Programming Languages 124

BIBLIOGRAPHY 128

LIST OF FIGURES

Figure 2.1	State Transitions Of A Memory Cell	17
Figure 3.1	Structure Of a EML Operator	31
Figure 3.2	Operator o.factorial	32
Figure 3.3	An EML program for merge_sort	41
Figure 3.4	Data Transfer Between Contexts	46
Figure 4.1	Operator o.sum_inc	77
Figure 4.2	Abstract form Of operator o.sum_inc	78
Figure 4.3	Operator o.integer_sum	79
Figure 4.4	Abstract Form Of operator o.integer_sum	80
Figure 4.5	Operator o.difference	83
Figure 4.6	Abstract Form of operator o.differnce	84
Figure 5.1	Operator o.quotient	90
Figure 5.2	Operator o.sub_and_add	91
Figure 5.3	A Program For Determining The Greatest Common Divisor	96
Figure 5.4	Operator o.gcd	97
Figure 5.5	Operator o.lgcm	100

Figure 5.6 Operator o.factorial 109

Figure 5.7 Operators Containing non_terminating Loops 111

Figure 6.1 Data Transfer between operators 123

LIST OF TABLES

Table 2.1: Channels Of Communication Between Computation And a Memory cell in EML 20

Table 3.1: EML entities and conventions for naming them 21

Table 3.2: Operators Of the EML Language 28

Table 3.3: Definitions Of the operators Of the EML Language 29

Table 4.1: Environments Of o.sum_inc(num,sum) 79

Table 4.2 Environments Of o.integer_sum(num) 82

Table 4.2A Environments Of o.integer_sum(num) during execution of the loop_statement 82

Table 4.3 Environments of o.difference(x,y) 85

Table 5.1 Environments of o.quotient(x,y) 90

Table 5.2 Environments of o.sub_and_add(x,y,q) 92

Table 5.3 Environments of o.gcd(x,y) 98

Table 5.4 Environments of o.lgcm(x,y) 101

Table 5.5 Environments of o.factorial 110

STATES, SIDE EFFECTS AND MULTIPLE CONTEXTS IN PROGRAMMING LANGUAGES

INTRODUCTION

Programming languages can be broadly grouped in two classes: (a) those having the concept of memory or store; and (b) those having no such concept. The applicative languages belong to the latter group while the von Neumann languages form the majority of languages in the former group. von Neumann languages have been noted for complex semantics, complex transition states and side effects. Programs in von Neumann languages are often difficult to prove or verify. The applicative languages, on the other hand, have simpler semantics, no notion of state and no side effect. Consequently it is often easier to prove or verify applicative programs (Backus '77). The notion of memory and state, however, are quite useful in programming applications which require history sensitive computation and which involve modeling and manipulating environments or objects acting in an environment. Data base systems and real time controller systems are examples of such applications. The investigations reported in this thesis have been motivated by the following question: Is it possible to design a programming language which retains the notion of memory and state AND yet has a semantics simple enough such that programs are easier to verify? This goal could be achieved if the programming language would satisfy the following:

(1) Within a procedure, it should be possible to assign different values to a variable at different stages of computation; i.e. it should be possible to write:

```
"  X := F1(Y);
      ...
   X := F2(Z); "
```

where X,Y,Z are names of variables
and F1,F2 represent procedures.

(2) The semantics of the language ought to be such that it should be possible to state the effect of executing a syntactically valid procedure as an expression.

The first condition assures that a variable may assume different values at different stages of computation thus allowing the state of the environment to be modified as computation proceeds. The second condition assures that the meaning of executing a syntactically valid procedure may be captured as a formal expression. This capability should ease the process of program verification. None of the current programming languages support both of these two conditions. The von Neumann languages allow assignment statements of the desired form. But the semantics of these languages are such that there is no assurance that the effect of executing a syntactically valid procedure in the language may be captured as an expression. In the applicative languages the procedures are really expressions; but they have no notion of variable, memory or state; they do not provide a facility for a variable to assume different values at different stages of computation. The data flow languages have the semantics of functional languages and yet they follow the syntax of imperative languages. These languages follow the "single assignment rule" i.e., they allow a variable to assume a value only once. Thus condition (1) stated earlier cannot be satisfied by the data flow languages.

It is difficult to state the effect of executing a von Neumann procedure as an expression because of side effects permitted by such languages. Side effects are a principal cause of the complex semantics of the von Neuman languages (Backus '77). In von Neumann languages side effects are practically indeterminate (see next chapter). Our principal hypothesis is that if side effects could be specified or restricted then the semantics of a programming language would be simplified and programs in such a language would be easier to verify. It is a common belief that programs which are easier to verify, are often easier to understand, maintain, and modify as well.

This thesis is organized in the following order. In the next chapter we analyze how side effects arise in von Neumann languages. From this analysis we propose a solution to restrict but not eliminate side effects. Notice that the elimination of side effects would also eliminate the notion of memory from a programming language and would be contrary to our goal. Next we describe the notion of a two-state memory system and a simplified virtual machine where the computation and memory communicate with each other via three unidirectional channels. These concepts are useful in developing the language E nvironment M odeling L anguage (EML) which enforces restricted side effects. The syntactic and semantic specifications of EML are described in chapters 3 and 4. In chapter 5 we describe an approach to verification of programs in the EML language by first transforming them to algebraic expressions. The final chapter tries to find a place for EML in the panoply of programming languages.

CHAPTER 2

CONSUMABLE VARIABLES & UNIDIRECTIONAL CHANNELS: MODEL OF A VIRTUAL MACHINE

2.1 How Side Effects Occur In von Neumann Languages:

In imperative languages side effects occur in three different ways:

(a) During the execution of a procedure, if values of one or more of its input parameters are altered then a side effect is said to have occurred.

(b) If during the execution of a procedure the values of one or more global variables are modified then the procedure is said to have side effects.

(c) During the execution of an assignment statement, if the values of one or more variables NOT occurring on the left hand side of the assignment operator are modified, then also side effect is said to have occurred.

We explain below how side effects occur in these situations.

Assume the following statement in a language like PASCAL:

"Y := F(X);"

During execution the procedure may alter the values of input variables if the parameters are passed by reference. The procedure may also access and alter the values of one or more global variables not referenced in the statement. Both of these cause side effects. In imperative languages there is no simple rule to detect and determine the effect of invoking a procedure. Not all procedures cause side effects, nor does every invocation of the same procedure cause side effects. In short side effects are allowed by the languages but the extent of side effect remains a property of each individual procedure and are difficult to control on a more global basis and certainly cannot be determined from the local program syntax. A language which wishes to control side effect must specify in the syntax which variables may be accessed by a procedure at a given invocation. Also the semantics of the language must specify what happens to the variables accessed by a procedure during execution. Such specifications must be a property of the language and must be applicable to all procedures in the language.

Imperative languages provide several mechanisms to bind more than one name to a single value at a given time. This happens when variables are passed by reference to procedures. In this case variable names in two different scopes continue to share a value. Another mechanism to bind more than one name to a value is to use pointers and refer to values via pointers. A third and more subtle form of binding multiple names to a value is inherent in the way components of structured objects are referenced and used. The value of a component of a structured object is bound to the name of the component as well as to the name of the parent object containing the component. In these situations, whenever the value of the component is altered, the value of the parent object is also altered, albeit indirectly. The side effects occurring from the latter are illustrated below. We shall see that side effects from these sources are more subtle and more difficult to control within the confines of the imperative languages.

Assume the following fragment of a 'C' program:

```
    Struct  rec
        {  int  A;
           int  B;}  X;
        /* X is a structure of two components A, B
           both of which are integers */

   {
        X.A = a;
        X.B = b;
          /* stage 1 */
          /* The components of X are initialized
                 to values a and b.*/
          ...
        X.B = X.B + b;
          /* value of X.B = 2b */
          ...
    }
```

At the end of stage 1, the value associated with the variable of name X is sequence of two values viz. (X = (a b)). In the last statement the value of the variable of name X.B is changed to 2b. However since X.B is a component of X, the reassignment of the value of X.B also indirectly changes the value of the structure X as a side effect. At the end of the reassignment of X.B the value of X is in reality a sequence of two values (X = (a 2b)).

Thus the value of X was changed although its name was not referenced in the statement. This form of side effect can become more subtle if the the component X.B is accessed via a pointer. If more than one pointer points to the component, then altering the value of the component via any of the pointers would cause a side effect to the structure containing the component as well as to the value accessed via the other pointers. Under these situations it is difficult to reason about the state of a structured variable. This analysis provides another clue for controlling side effects. A language wishing to control side effects must make sure that a value is not accessed via more than one name. In the above example, the component B is accessible via two names X and X.B. If the value associated with the name X.B is changed, side effects is caused to the variable X. If we could split the component B from the structure X, before reassigning its value, then we would be able to prevent the side effect on X. This however requires that the language be able to decompose (and compose) a structured data during execution. The von Neumann languages do not provide any such mechanism.

What causes side effects in imperative languages and why is it harmful? As discussed above, side effects can arise only because the imperative languages allow a value to be shared (or bound) among more than one name at a time and also because these languages allow name spaces of procedures to be overlapped. For example, the global variables are included in the name space of all procedures in a program. These languages allow the sharing to occur but does have no mechanism to reflect which names are sharing a given value at a given stage of computation.

Side effects are not harmful par se except that side effects make it difficult to reason about the state of a program. For program verification it would be necessary to be able to reason about values associated with names. If values are shared then it will be necessary to keep track which names share which value(s) at each stage of computation, so that when a value is modified, the states of all names sharing the value may be updated. This is cumbersome at the least and may be almost impossible in many cases if the language allows the value to be referenced indirectly. Because of side effects, the effect of executing an assignment statement cannot be stated by means of a simple rule. Nor can the effect of executing a procedure be captured by any simple rule because of side effects.

2.2 How Side Effects Could Be Controlled In von_Neumann Languages:

Our aim is to design a programming language which retains the notion of state and memory and yet it should be easy to state the effect of executing a program in the form of an expression. For this purpose it is necessary to make sure we are able to express the effect of executing a statement in the form of an expression from the statement itself. From our analysis of side effects and how they occur, we conclude that to meet the above goals, a language ought to meet the following requirements:

(1) It should be possible to perform multiple assignments in one step.

(2) The convention for naming values should be such that a value is never attached to more than one name at a time.

(3) The name spaces of procedures must never overlap.

(4) The strategy for passing parameters to a procedure should be such that a procedure may be viewed as a function and yet it is able to alter the environment in which it is invoked.

(5) The loop statement should be so constructed that it is possible to easily deduce the function computed by a loop statement.

The first requirement is easily met. The normal model of assignment in imperative languages is that a procedure or a function may accept multiple parameters but may return a single value. The desired language should allow a vector of values to be accepted and a vector of values to be returned by a function. The difference in the two models are shown below.

```
Model of assignment in imperative languages:
X := F(Y1,Y2 ...Y_n);

Model of assignment in EML:
F(Y1, Y2..Y_n) = (X1,X2...X_m);
```

We have not found a simple way to accommodate the second, third and fourth requirements within the framework of von Neumann languages. These requirements force a fundamental modification in the way a programming language views and references its data. This paper describes the design considerations, syntax and semantics of the programming language EML which satisfies the requirements stated above.

The features of EML may be summarized as follows. It has a very simple flow of control. It allows procedures to return a vector of values rather than a single value. It makes sure that a value is always bound to a single name at a time although the value may be bound to different names at different times. It makes sure that a value is always viewed as a scalar and never as a component of another value. To support this view it provides operators to aggregate and split structures during execution. Further EML makes sure that the effects of executing a statement are inferred from the statement itself. For this purpose it observes the following rules:

(1) Specify, in the syntax, which nonlocal variables a procedure may access at a given invocation. One implementation of this rule would be to restrict the access of a procedure to only those variables in nonlocal memory which are passed to it as parameters. A procedure may not access any other variable in the nonlocal memory. This rule would constrain the possibility of unbridled changes

in global memory.

(2) Specify, in the semantics of the language, what happens to a variable in the nonlocal memory if it is accessed by a procedure during execution. This specification will be a property of the language rather than of the individual procedure. Thus the effect of accessing a nonlocal variable will be independent of the internal operation of the procedure.

2.3 Consumable Variables:

The above rules form the core of the programming language Environment Modeling Language (EML) described in this thesis. While the implementation of rule (1) is quite clear cut, there are probably many ways in which the second rule could be enforced. EML uses the concept of 'consumable variable' to implement the second rule. In EML, a variable is a value stored in a cell in the memory. The value can be completely removed from the cell which merely serves as a container of the value. If the value is removed, the cell becomes empty; i.e it contains a special value empty designated as "E." "E" is a legal value of all variables. If a variable 'X' is passed to a procedure P, then the value of X is supposed to be consumed by the P; i.e., the procedure removes the value from the container and consumes it in computation. Specifically, the following is supposed to happen when the variable X is passed to the procedure P. The value 'x' is removed from the container labeled 'X'. Thus the container 'X' becomes empty. The value 'x' is then placed in a container local to scope of the procedure 'P'. This concept of consumable variables has two interesting implications:

(1) A variable always has the value 'E' after it has been made available to a procedure, no matter what the procedure does.

(2) This scheme assures that conceptually a procedure will only access local values. This enables us to view the procedure as a function and in turn helps to capture the effect of executing a procedure in terms of an algebraic expression.

Other properties and rules of usage of variables will be described later. It suffices here to note that, with respect to the second rule, EML specifies that after a variable has been used by a procedure, it must contain the specific value 'E'. Further, a variable containing value 'E' may not be accessed for consumption by any other procedure. While the notion of an empty variable is similar to the concept of unbound variables in some languages like PROLOG, the notion of consumption of value does not seem to have a parallel in other programming languages.

It is interesting to compare our solution for controlling side effects with the solution adopted in the data flow languages (Agarwala '82, Ackerman '82, Arvind '82, McGraw '82). The data flow languages do not allow any side effects although some allow assignment statements. These languages have no notion of global memory.

With respect to the second rule above, these languages implement the policy of 'no change'; i.e the languages specify that the parameters passed to procedures or functions may not change in any way. This is enforced by the convention that all parameters are passed by value only. In contrast, EML may be said to implement the policy of 'pro change'; it requires that parameters passed to procedures must change and specifies the nature of that change, from full to empty. The policy followed by data flow languages eliminates side effects completely. Execution of a procedure in these languages does not affect its environment at all. So far as a data flow procedure is concerned, environment does not even exist. EML procedures on the other hand recognize that there exists an environment in which the procedure executes and execution of a procedure causes specifiable changes in the environment. In order to support the consumable data model of EML, we have found it helpful to modify the traditional view of virtual machines modeled by von Neumann languages in two different ways. The EML virtual machine is a von Neumann machine modified as described below.

2.4 Notion Of A Two-state Memory System:

The von Neumann languages traditionally view memory as a series of cells; the cells are used to hold values of variables. A cell always contains some value, whether or not the value is of any significance to a program manipulating the cell. In these languages a cell may NEVER be devoid of a value. The state of a cell is altered by assigning a new value to it. Thus conceptually a cell in the memory may be said to exist in only one state viz. the 'filled' state , designated as (F). In these systems there is only one possible type of state transition for a memory cell. The state of a cell may only be changed from one filled state Fi to another filled state Fj (Fig 2.1). We will call this type of transition a 'filled_to_filled' transition.

The memory cells in EML are permitted to have two types of state transitions called consume transition and produce transition. These transitions relate to the syntactic constructs consume expression and produce expression described in the syntax of the language (Chapter 3). The consume transition occurs when a variable is passed as a parameter to a procedure. At the beginning of the consume transition the memory cell must be full, i.e., the variable representing the cell must have a value other than 'E'. At the end of the consume transition the cell must contain the value 'E'. Thus the consume transition of cell is indicated by '(Fi) ---> (E)' where Fi <> 'E'. The produce transition occurs when values are assigned to memory cells. At the start of the produce transition a cell may have any value including 'E'. If at the beginning of produce transition a cell has any value other than 'E', then the cell is first emptied and then filled with the new value. Thus at the end of the produce transition a cell must have a value other than 'E'. Thus the produce transition is indicated by '(Fi) ---> (Fj)' where Fj <> E. The produce transition is roughly similar to the state transition of cells in the von Neumann languages. These languages do not have any notion equivalent to the consume transition proposed here. The permissible state transitions of cells in EML memory are shown in figure 2.1. The notion of the memory system proposed here may be compared with the notion of memory of other languages in the following manner. In EML, memory is writable and there are two types of read operations: destructive and nondestructive. The latter variety of read operation implements the consume transition. In von Neumann languages, memory is readable and writable; there is no concept of destructive read. In data flow languages, memory is readable but a memory location can be written to only once. This restriction on write operation implements the single_assignment rule which forms the core of data flow languages. The functional languages are at the other end of the spectrum since they have no notion of memory at all.

Figure 2.1
State Transitions Of A Memory Cell

```
(Fi) --> (Fj)              (E)---->(F)

 Fig 2.1a                   Fig 2.1b


Consume transition   (Fi) ---> (E)    (Fi <> E)
Produce transition   (Fi) ---> (Fj)   (Fj <> E)

            Fig  2.1c
```

2.5 Unidirectional Channels Linking Computation And Memory:

In the traditional view of the von Neumann languages, the communication between the cpu and memory is performed via a single bidirectional channel. Consider a simple statement in a language like 'C':

```
if ( x > 5) x = f(x);
```

In this statement the location x is used in three different ways, once for comparing the resident value with another constant, once for using the value as an operand to the function f and finally the same location is used for storing the newly computed value 'f(x)'. Logically there are three different purposes for accessing a memory cell during program execution.

(a) to 'look-up' and learn some thing about the property of the value resident in the cell;

(b) to use the resident value in some computation to generate a new value;

(c) to store a new value to the cell.

In two of these instances the data value flows from the cell to the computation; in the third instance the data value flows from the computation to the cell. All data transfer between computation and the cell is performed via only one channel no matter what the purpose of access is or direction in which the data flows. The semantics of accessing a memory cell are not reflected in the machine structure.

In the EML system, the communication between the cpu and the memory cell is assumed to be performed via three functionally specialized unidirectional channels. Each channel is dedicated to a specific purpose. These three channels are : the image-channel, the consume-channel and the produce-channel. The properties of these channels are stated in table 1. When accessed via the image-channel, an image of the value resident in the cell is transported to the cpu; the cell itself remains unchanged. When accessed via the consume channel, the value resident at the cell is transported to computation (i.e., the cpu). The cell itself reaches an empty state. If the target cell was already empty, no data transfer occurs and an error results. Any instruction causing such an error is not executed. When accessed via the produce channel value flows from the cpu to the cell and the receiving cell becomes filled with the value destroying the old value. The value 'E' cannot be transported by the consume or produce channels.

2.6 EML-machine vs. Other Virtual Machines:

In von Neumann languages, a value may never be completely dissociated from the cell containing the value. A cell may never be emptied by any means. A value assigned to a cell destroys its existing value. The data flow languages, on the other hand, impose the 'single assignment rule' i.e a value may be assigned to a variable only once. Once assigned, a variable may not be reassigned new values ever again within the scope of definition of the variable (Ackerman '82). In our view, this implies that in the data flow languages the only type of state transition allowed is from "E --> F"; here E represents the value in the variable before the first value was assigned to it by the program. In these languages, the filled state is a dead end for a memory cell. Once a cell is in a filled state no other transition is allowed. In contrast, in the EML language, a value can be completely dissociated from the container; a full container may be emptied by removing its value and passing it on to a procedure; an empty cell may not be emptied again; and any new value except "E" may be loaded to any cell.

Table 2.1
Channels of communication between computation and a memory cell in EML

channel name	direction of data transfer	effect on state of the memory cell	when used
image_channel	cell ---> computation	state unchanged	condition_expression.
consume_channel	cell ---> computation	(Fi) ---> E cell emptied	consume_transition
produce_channel	computation ---> cell	(Fi)----> (Fj) empty cell	produce_transition.

CHAPTER 3

SYNTACTIC SPECIFICATION OF EML

3.1 EML-entities:

The EML language consists of the following type of entities: constants, variables, templates, and operators. Files and operations related to input/output are not included in the current definition. A EML program or a user-defined operator is made up of statements composed of the above entities along with other user-defined operators. The conventions for naming user defined entities are shown in table 3.1.

Table 3.1
EML entities and conventions for naming them.

EML entity	Prefix for name
constant	c.
template	t.
variable	v.
file	f.
operator	o.

CONSTANTS: The constants include integers, reals, characters and other user defined constants; names of all user-defined constants start with the prefix "c." User defined constants are declared as:

```
define (20 ) =      c.twenty;
define ( c.twenty * 5)  =   c.hundred;
define ("catalog")  =    c.catalog;
```

In the above c.twenty is declared to be a constant of integer value 20 and c.catalog is defined to be a string constant of value "catalog".

TEMPLATES: Template entities are descriptors of forms. There are five fundamental forms known in the language. They are : integer, real, boolean, character, and sequence. The first four forms resemble the principal types in other languages. The form sequence is obtained by concatenating one or more of the principal forms. Sequences are also formed by concatenating one or more sequences. Further a sequence may be split by the split operator at any indicated location to generate two sequences. The "split" and "concat" operators used for these operations are described later. A sequence must have one or more component forms. In fact the fundamental forms are each considered as a sequence of one component. Attempt to split an empty sequence causes an error. Some examples for declaring sequences are shown below:

```
(1)  sequence(integer real boolean character)
                   =    t.all;
(2)  sequence ((integer 5) (real 1) (char 20))
                   =   t.record;
(3)  sequence ( t.record 10)  =   t.ten_record;
```

Example (1) indicates that the template t.all is obtained by concatenating the four fundamental forms in the order described. The template t.record is a sequence of three sequences: the first one is a sequence of five integer forms; the second one contains one real form and third one is a sequence of twenty forms of character. The template t.ten_record is a sequence of ten forms of the template t.record. Template definition is a mechanism for defining a storage structure without binding any variable to the form. Note that this mechanism of concatenating objects suffices for arrays, lists and record structures. The template definitions should be particularly useful in input/output statements.

Constants and templates do not occupy any cell in the memory. They merely form declarations for the ease of compilation.

VARIABLES: Variables are containers of values. Names of all variables must start with the prefix "v."; valid variable names are v.var, v.var_x, v.var1 etc. Variables may be of simple form, or be of a complex form defined by a template. The model of variables of EML differs significantly from the model of variables in other imperative languages. Most of the significant differences will be presented in the section on rules for usage of variables. In this section we explain the EML view of variables as scalar objects. No matter how complex the structure, i.e., template, of a variable is, EML treats all variables as scalar objects; i.e. variables are passed between operators as single entities. In languages like PASCAL, the fields of a record are named by associating the field name and the name of the record by the dot ('.') operator. Other languages provide similar mechanisms. In these languages different procedures may directly access and simultaneously process different parts of a record. Since even variables of complex structures are treated as scalars, in EML there is no provision to directly access or name a specific component of a variable. To access a component of a variable, first the variable must be processed to isolate the desired component as an independent variable. An example of such operation is given below:

```
make_sequence (0,10) = v.num_array;
split (v.num_array, 5)  = (v.upper,  v.lower);
split (v.upper,4)   = (v.first_four,  v.fifth_num);
```

The first line is an instruction to create a sequence of 10 elements each of which is initialized to the value 0. The sequence is associated to the name v.num_array. Thus num_array is essentially equivalent to an array of ten integers in other languages. We intend to obtain the fifth element of this array. The split operator splits a concatenation of objects in two parts. 'make_sequence' and 'split' are built-in operators of the EML language. More precise definition of 'split' and other EML operators and definition of EML statements will follow shortly. Since the variable num_array is viewed as one object, it is not possible to access the fifth element by simply subscripting the array with the desired index. Rather, the variable is first split in two parts after the fifth element producing two objects corresponding to the upper and lower halves of the original object. The upper half is then split again after the fourth element to generate the variable "fifth" as an independent variable. The variable "fifth" may now be used as a scalar object and passed to any operator. The decision to treat all variables as scalars, independent of the complexity of its structure has two significant implications:

(a) it requires that the language provide operators to rearrange structures of variables at runtime ; the EML provided operators "split" and "concat" serve this need (see built in operators of EML language).

(b) since structures of data are necessarily altered while accessing any of its components, the notion of static typing akin to PASCAL are less useful in this language.

Other advantage of treating complex data as scalars lies in controlling side effects. This aspect has been discussed by Ackerman (Ackerman '82).

EML_STATEMENTS:

The EML_statements are analogous to instructions in other programming languages. They contain operators, variables, constants and the equality sign. More will be said about the EML_operators below. Here we describe the structure of the EML_statements. There are three types of statements: simple statement, conditional statement and the loop statement. Their general syntax is shown below:

```
F(X) = Y  ;      simple statement.
if (eq(X,Y)) F1(X) = Y;   conditional statement.

loop
[ if (gt (X,Y)) sub (X,1) = X; ]
```

The first statement merely indicates, apply the EML operator F on X and store the resultant value in the variable Y. This looks familiar to the assignment statement of imperative languages. The difference is that on execution of the statement the value of X will be consumed, i.e. after the statement is executed the value of X would be empty (E). The conditional statements have a condition expression associated with them. Loop statements are preceded by the key word 'loop'. Syntactically speaking a statement has two components: the condition expression and the procedure expression ; the condition expression is optional. The procedure expression consists of a consume expression and a produce expression. If it exists, the condition expression starts with the key word "if". The condition expression binds two expressions via a relational or boolean operator to generate a truth value. The operands in the condition expression may be variables, constants, arithmetic or boolean expressions involving these entities. Only relational, boolean and arithmetic operators defined in the EML language are allowed in a condition expression. User-defined operators may not be used in condition expression. User-defined operators in general return a vector of values rather than a single value. It is not always simple to obtain a single_value from such functions. Hence in the first definition of the language user_defined operators have been excluded from condition expression. A consume expression contains a user-defined operator or a built-in operator of the language and one or more parameters; a produce expression is simply a list of of one or more variable or file objects. The abstract form of an EML-statement is shown below:

```
 if (c_i (X1, c_j(X2,X3)))
    f_I (X1, X2.... X_n)
        =
        Y1,Y2.....Y_n;
```

In this example first line represents the condition expression, the second line represents a consume expression and the last line represents a produce expression. c_i, c_j etc. are operators of the EML language. f_i is either a built-in or user defined operator. X1,X2...Xn are either variable names or constants or file names. Y1,Y2...Yn must be variable names or file-names.

EML-OPERATORS:

The operators in EML are the active units; they process data items to generate new data. The EML-language supports a number of built in operators (Table 3.2). Table 3.3 shows the definition of these operators. A program in EML is actually an user-defined operator. A user defined operator is composed from other user-defined operators and/or built-in operators of the EML language. Names of all user-defined operators start with the prefix "o.". In EML, operators are used in two ways: they may be used in condition expressions or consume expressions with fairly different results. In either case, operators act on their operands to generate new values. When used in a condition expression, an operator behaves like a function in algebra; i.e execution of an operator in the condition expression does not alter the operands themselves. Further, new values generated are not stored in memory; they are merely used to compute a truth value. On the other hand, when used in a consume expression, operands supplied to an operator are consumed during execution of the operator; new values produced are stored in the memory. These aspects are fully explored in the section on semantic specification of the EML language.

Table 3.2

Operators of the EML language

Class	Operator
Relational	lt,gt,eq,le,ge, ne
Boolean	and, or, not
Arithmetic	add,sub,mult,div , idiv, mod
Structure rearrangement	split, concat
Locational	locate
object transfer	receive. send
fill	copy, mk_sequence

Table 3.3
Definitions of operators of the EML language

```
-----------------------------------------------
  (a) relational operators:
   lt (X1, X2 )  :: if ( X1 < X2)
                    then true else false;
   gt ( X1,X2 )  :: if ( X1 > X2)
                    then true else false;
   eq ( X1,X2 )  :: if ( X1 = X2)
                    then true else false;
   ge ( X1,X2)   :: if ( X1 < X2)
                    then false else true;
   le ( X1,X2)   :: if ( X1 > X2)
                    then false else true;
   ne ( X1, X2)  :: if ( X1 = X2)
                    then false else true;

  (b) boolean operators:
   and ( X1, X2 ) :: if ( X1) and if (X2)
                     then true else false;
   or (X1, X2) :: if ( X1) or if (X2)
                     then true else false;
   not ( X1) :: if (X1) then false else true;

  (c) arithmetic operators:
  add (X1, X2 ) ::    ( x1+x2);
  sub ( X1, X2) ::    ( x1-x2);
  mult ( X1, X2) ::   ( x1*x2);
  div ( X1, X2) ::    ( x1/x2);
  idiv ( X1, X2) ::   ( quotient (x1/x2))
  mod ( X1, X2) ::    ( remainder ( x1/x2))

  (d) locational operators :
  locate ( (a1,a2,.....a_n), i) ::  (a_i);


 (f) structure rearrangement operator:
   split( (a1, a2, ...a_n ),   i )  ::
     return ( (a1,a2...a_i) ,(a_i+1, , , a_n) );

   concat ( (a1,a2...a_j), (a_k, .. a_n)) ::
     return( a1,a2....a_j,a_k,...a_n);
-----------------------------------------------
```

3.2 EML Programs / User-defined Operators:

Names of all user defined operators start with the prefix "o.". A syntactic description of the structure of user defined operators is shown in figure 3.1. An EML operator contains two sections: the declarative part and executable part. All variables used in an operator must be declared in the declarative part. If user-defined constants are used they must be also be declared here. Optionally, user-defined templates and associations between variables and templates may also be contained in this section. The declarative part of a EML operator is similar to the declarations in a PASCAL procedure, except that it is not obligatory to associate types with variables. An EML programs to determine factorial is shown in figure 3.2. An EML program implementing merge_sort is presented at the end of this chapter (figure 3.3).

Figure 3.1

Structure of a EML operator.

```
EML_operator :: <declarative> <executables>;
declaratives :: <template_def><constant_def>;
executables :: <receive> [<body>] <send>;
receive  :: "receive"  "()"
            "="  <object_list> ';';
send :: "send" <parameter_list> "." ;
body :: <statement> | (<body> <statement>;
statement ::<procedure_expression>
             | < cond_statement>
               | <loop_statement> ";" ;
cond_statement :: <cond_expression>
               < procedure_expression>;
loop_statement :: "Loop" "["(<cond_statement>
              | <procedure_expression>) "];
cond_expression :: "if" "(" <relational_expr>
               | <boolean_expr>')';
procedure_expression::
              <consume_expression>
               '='
              <produce_expression> ';' ;
consume_expression:: <operator_name>
            '(' <parameter_list> ')';
produce_expression:: <object_list>;
parameter_list :: <objects>   | constant |
                 ( <parameter_list>
                   <object> <constants>);
objects :: <variable> | <file>;
object_list :: <objects>
          |( <object_list> ',' <objects>);
```

The executable part of a EML operator contains a body braced between a receive statement and a send statement. The body itself is optional. If it exists it is a sequence of one or more statements. The structure of statements have been described above.

Figure 3.2

Operator o.factorial

```
o.factorial
{ vars : v.num,  v.num2, v.nextnum,
        v. factorial,  v.nextfact;

  receive ()  = v.num;
                                /* # 0 */
  if (eq( v.num,0))
     add(v.num,1) =  v.factorial;
                                /* #1  */
  copy ( v.num,2) = (v.num, v.num2);
                               /* #2  */
  sub ( v.num, 1) = v.nextnum; /* #3  */
  o.factorial ( v.nextnum) = v.nextfact;
                               /* #4 */
  mult ( v.nextfact, v.num2) = v.factorial;
                               /* #5 */
  send (v.factorial).           /* #6 */
}
```

The first statement in the operator o.factorial is the <u>receive</u> statement. The operator is to receive a value and store it in the variable v.num. If the value of the variable is 0, then 1 is added to the value of variable and the resulting value is associated with the variable v.factorial. This happens in the second statement. The third statement attempts to copy the value of v.num to v.num and v.num2. Note that the statements #2 to #5 can execute only if the statement #1 did not execute. If the statement #1 did execute then the variable v.num would be <u>empty</u>. As will be stressed throughout this paper, a statement containing an <u>empty</u> variable cannot execute. Thus if the statement #1 would execute, then the statements #2 to #5 would not execute. On the other hand if the value of v.num would be greater than 0, then the statement

#1 would not execute. The variable v.num would not be <u>empty</u> and therefore the statements #2 to #5 would execute. In any event the <u>send</u> statement would execute.

3.3 Rules Of Usage Of Various Entities:

It is useful to have a general idea of the semantic model of the EML machine before we describe the syntactic restrictions on various constructs of the EML language. The complete description of the semantic model of this language is the subject of the next chapter. Here we will present a brief and informal overview. It has already been stated that operators may be present in both <u>condition expressions</u> and <u>procedure expressions</u>. Execution of a <u>procedure expression</u> empties the variables in <u>consume expression</u> and fills new values in the variables in the corresponding <u>produce expression</u>. In other words, execution of an operator in a <u>procedure expression</u> reduces the consumed operands to the state "E" (for empty) and transfers the produced operands to the state "F" (filled). Further, a <u>procedure expression</u> can be executed only if the corresponding "readiness condition" is satisfied. The readiness condition states that an operator in a <u>procedure expression</u> may be evaluated only if none of the consume operands are in the empty state. If the readiness condition fails then the EML machine ignores the statement containing such a <u>procedure expression</u> and proceeds to examine the next statement in the body of the operator.

Furthermore, while values of variables are consumable, values of constants and files are not.

3.3.1 Use Of 'receive' And 'send' Operators:

The receive operator may occur only in the consume expression of the receive statement; the send operator may occur only in the consume expression of the send statement. They may not occur in any other statement in the body of an operator. The receive statement must be the first executable statement and the send operator is the last executable statement of an operator. These statements delimit the body of an operator; in this sense they serve like the "begin-end" constructs of PASCAL. However, in EML their main function is to convey values to and from the calling operator. The structure of receive and send statements are shown below:

```
receive ()   =  X1,X2,...Xn;
send ( Y1,Y2,...Ym).
```

The first line indicates that the operator is to receive some values from the calling operator ; these values are to be placed in the variables X1,X2..Xn.

The process of transporting values between calling and called operator may be viewed as described in figure 3.4. Assume the operator "A" calls the operator "B" through the i_th statement in its body: "B(X) = Y". Also assume that the receive and send statements in operator "B" are: "receive() = M" and "send (P)" respectively. In the semantic model, the calling and called operators are supposed to be connected via two unidirectional pipes. The receiving-pipe transports value(s) from the calling operator to the called operator and the sending-pipe transports value(s) from the called operator to the calling operator. There is no other means of communication between the calling and called operator. Assume that when the operator "B" was called the variable X in "A" contained the value "xi". The following steps happen when the operator B is invoked within the operator "A".

(1) the value xi contained in the cell X is dissociated from the cell itself; the result is an empty container X and the dissociated value "xi".

(2) The value xi is sent down the receiving pipe to the called operator B.

(3) In B, the <u>receive</u> operator accepts the value xi and places it in the empty container M in B.

The operator B terminates when it executes its <u>send</u> statement. The value contained in its variable P is then shipped to the calling procedure A in the following steps.

(1) The value p_n contained in P is dissociated from the cell P. It produces an empty cell P and dissociated value p_n.

(2) The value p_n is sent down the <u>send</u> pipe to the environment in A.

(3) The value p_n is accepted by A and placed in the cell corresponding to the <u>produce</u> <u>expression</u> of the statement which called B. The net result is that the value p_n

is placed in the cell Y in the operator A.

The consume expression of the receive statement may not contain any variable. This is obvious since the only purpose of the receive operator is to accept values from the external environment. The consume expression of a send statement may or may not contain an operand. If it contains some variables, as is the case in the example above, it indicates that values contained in the operands will be returned to the calling operator. It may be noted however, that if in an operator, the produce expression in the receive statement is an empty list and there are no operands to the send operator, then the operator neither receives any value nor returns any value to the calling operator. Execution of such an operator is equivalent to performing a "no operation" instruction.

3.3.2 Use Of Operators:

Operators may appear only in condition expressions and consume expressions. Produce expressions may not contain any operator. A consume expression may contain only one operator. Since all operands in the consume expression must be named objects, nesting of operators in the consume expression cannot be supported. Execution of nested operators would produce values without any name. In such cases the semantic analysis of an operator becomes quite difficult under our specification (see chapter 4). There may be more than one operator in the condition expression.

Following is a valid use of operators:

```
if ( gt ( add(X,Y), sub ( Z,Y)))
        o.F( X, Y) = (Y, Z, P);
```

In this example there is nesting of operators in the _condition expression_. Following two statements are examples of invalid uses of operators:

```
o.F(X, o.F1(Y,Z)) = (P,Q);
o.F(X,Y) = o.F1( P,Q);
```

In the first example there is nesting of operators in the _consume expression_. In the second example the operator F1 appears in the _produce expression_.

3.3.3 Use Of Variables:

(1) Variables may appear in _condition expression_, _consume expression_, _produce expressions_.

(2) Within a _condition expression_, a specific variable may be referenced more than once.

(3) Within a _consume expression_, a specific variable may appear only once.

(4) Within a _produce expression_, a specific variable may appear only once.

3.3.4 Use Of Constants:

Constants may be used in _condition expressions_ and _consume expressions_. They may not be used in _produce expression_. Within a _consume expression_ , a constant may be referred to more than once since constants are not consumable items in EML.

3.4 Loop Statement:

Looping requires that a sequence of statements be executed repeatedly. Moreover, it requires that a variable, say X, be capable of assuming new values on each iteration of the loop. This in turn requires that the model of memory ought to support filled-to-filled transition of cells in memory as described in chapter 2. Since this is the model of memory in von Neumann languages, such languages capture the semantics of looping in very simple syntax. The notion of writable memory also allows a simple syntax to capture the idea behind a loop statement. The notion of single_assignment of the data flow languages is basically incompatible with the semantics of looping, since the latter would require that the variable X be assigned a value at each iteration. In a pure form these languages cannot express looping. Looping in these languages, VAL and Id for example, are expressed through constructs like "new" which indicate that at each iteration a new instance of the variable X is being assigned the value (Ackerman '82, McGraw '82).

In EML, the general form of a loop statement is as follows:

```
 loop
 [ { C_i } f_i( X1, X2, ...Xn) ---> (X1,X2,....Xn); ]
```

The following syntactic restriction apply for the loop statement:

The set of objects in the <u>produce</u> <u>expression</u> must be a identical to the set of objects in the consume expression of the loop statement; i.e. the variables consumed by the operator must also be produced by the operator.

The necessity for such a restriction may be understood as follows: assume the variable Xi is present in the consume expression but not in the produce expression of a loop statement. In this case in the first iteration through the loop, the value in Xi will be consumed; the value in Xi will not be regenerated since Xi does not appear in the produce expression. Hence the readiness condition (which requires that none of the consume operands be in the empty state) will fail. As a result, the loop will not iterate beyond first pass. The semantics of the looping operation requires that the iterations terminate only when the condition C_i fail ; looping ought to continue otherwise. The syntactic restriction imposed here is designed to attain this objective.

Informally, execution of a loop_statement may be viewed as a two step activity.

```
while ( C_i = true)
{ 1. execute the operator f_i with values in the
     variables X1,X2 etc.
     As a result the variables X1 etc are emptied;
     i.e they are associated with the value E.
   2. Fill the variables in the procedure expression
     with values returned by the operator executing f_i.
}
```

In the first step the consume variables are emptied. In the second step the produce variables are filled with new values. Since in the loop_statement they are the same variables, with each iteration in the loop, the <u>consume variables</u> are associated with new values.

Note that if the loop statement does not have any condition associated with it, the result is a nonterminating loop. The loop is also nonterminating if there is no overlap between the set of variables in the <u>consume expression</u> and set of variables in the <u>condition expression</u>. In the latter case, alterations in the values of variables in the <u>consume</u> and <u>produce</u> expression will not affect the truth value of the <u>condition expression</u>. In this situation either the loop will not execute at all, since at the beginning of first iteration the condition will fail; or it will never stop since the loop condition will never fail.

Figure 3.3
An EML program for merge-sort.

```
o.merge_sort
  /* accepts a sequence and size of sequence;
     produces a sorted sequence */
 {
   receive () = (v.list, v.size);
   if ( ge(v.size,2))
        o.divide_and_merge (v.list,v.size)
               = (v.list,v.size);
   send (v.list, v.size).
 }

  /* Operator o.divide_and_merge:
     this operator accepts a list and its size; it
     splits the list in two halves, sorts the
     halves and merges the sorted halves to
     produce the sorted list.
  */
o.divide_and_merge
{
  receive() = (v.list, v.size);
  o.split_half(v.list,v.size)
           = ( v.left_half, v.left_size,
                 v.right_half, v.right_size);

 o.merge_sort( v.left_half, v.left_size)
         = (v.sorted_left, v.left_size);
 o.merge_sort (v.right_half, v.right_size)
         = (v.sorted_right, v.right_size);

 o.merge (v.sorted_left, v.left_size,
           v.sorted_right, v.right_size)
         = (v.sorted_list, v.size);

 send( v.sorted_list, v.size).
}
```

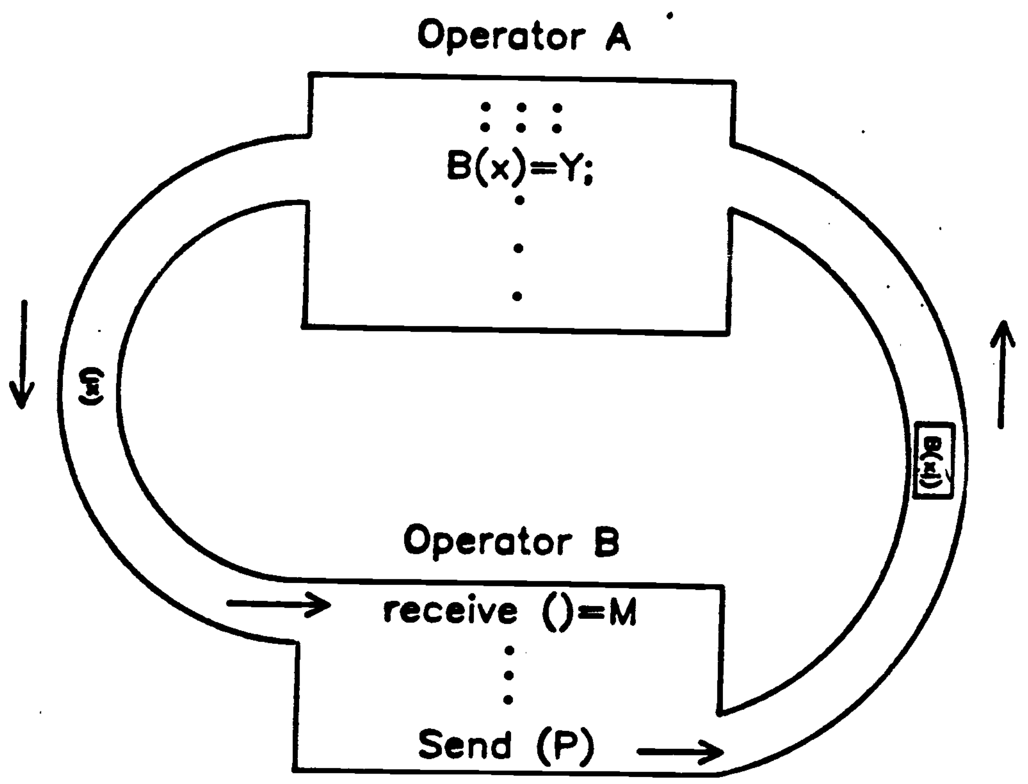
Figure 3.3 (continued)

```
 /* Operator o.split_half:
    this operator will split a sequence in to
    two halves. It will return the two halves
    and the size of individual halves.
  */
o.split_half
{
   receive () = (v.list, v.size);
   copy( v.size,2) = (v.size1, v.size2);
   divide (v.size1, 2) = v.half;
   copy(v.half,3)
        = (v.half1, v.half2, v.left_size);
   sub (v.size2, v.half2) = v.right_size;
   split( v.list, v.half1)
        = (v.left_half, v.right_half);
   send( v.left_half, v.left_size,
         v.right_half, v.right_size).
}


 /* Operator o.merge
    merges two sorted lists into one sorted list.
  */
o.merge
{
  receive ()
        = (v.list1, v.length1,
             v.list2,v.length2);
  copy(v.length1,2) = (v.length1, v.len1);
  copy(v.length2,2) = (v.length2, v.len2);
  copy(-1, 1) = (v.list_collector);
  /* this one element sequence acts as the seed on
     the sorted elements are collected.
   */
  loop
  [ if (and(ge(v.len1,1), ge(v.len2,1)))
        pick_sort(v.list_collector, v.list1, v.len1,
                  v.list2, v.len2)
         = (v.list_collector, v.list1, v.len1,
                  v.list2, v.len2); ]
   /* merge the residue of the longer list to
        the list collected so far. */

   if (gt(v.len1,0))
        concat (v.list_collector, v.list1)
               =  (v.sorted_list);
   if (gt(v.len2,0))
        concat (v.list_collector, v.list2)
               =  (v.sorted_list);

   add(v.length1, v.length2) = (v.length);
   /* remove the first element form the sorted_list */
   split ( v.sorted_list,1)
               = (v.list_collector, v.sorted_list);
   send( v.sorted_list, v.length).
}
```

Figure 3.3 (continued)

```
 /* Operator pick_sort:
    This operator accepts two lists, their lengths
    and a collector list. It detaches the first
    element in each list and cocatenates them on the
    collector in a sorted order.
  */
o.pick_sort
{
 receive() = (v.collector, v.list1,v.length1,
                 v.list2,v.length2);

 split( v.list1,1) = (v.head1, v.body1);
 split (v.list2,1) = (v.head2, v.body2);

 if (le(v.head1, v.head2))
   pick_merge (v.collector, v.head1,
                 v.body1, v.length1,
                 v.head2, v.body2, v.length2)
      = (v.collector, v.next_list1,
          v.next_length1, v.next_list2,
         v.next_length2);
/* else  */
pick_merge (v.collector, v.head2,
             v.body2, v.length2,
             v.head1, v.body1, v.length1)
          = (v.collector, v.next_list2,
              v.next_length2, v.next_list1,
              v.next_length1);

send( v.collector,v.next_list1,
        v.next_length1, v.next_list2,
        v.next_length2).
}


  /* operator o.pick_merge
     accepts a collector_list and descriptions of
     head, body, size of two lists add_head and
     other_head. It adds the add_head to the
     collector and decreases the
     its size; also it composes the other list.
     Returns all the lists .
   */
o.pick_merge
{   receive ()
       = (v.collector, v.add_head,v.add_body,
           v.add_size, v.other_head, v.other_body,
           v.other_size);
    /* concat v.add_head to the collector list and decrease
       the size of the v.add_size.
     */
   concat (v.collector, v.add_head)
               = (v.sorted_list);
   sub ( v.add_size, 1) = v.next_size;
   /* reconstruct the other list */
   concat ( v.other_head, v.other_body)
               = v.other_body;
   send (v.sorted_list, v.add_body, v.next_size,
               v.other_body, v.other_size).
}
```

Fig 3.4. Data Transfer Between Contexts



CHAPTER 4

SEMANTIC SPECIFICATION OF EML

4.1 Overview :

The target of semantic specification is to formally describe what happens when the EML machine is executing a syntactically valid EML program. Our aim is to develop expressions which represent the working of the EML machine. Also we shall show how, with the help of these expressions, one can specify the meaning of executing an EML operator. We will first define some terms useful for the rest of the discussion in this chapter. Then we will describe, informally, the working of the EML machine. Later we shall define some functions to formalize the concepts described in the informal overview. These functions will be used to formally specify the meaning of executing a statement and an EML operator.

4.2 Definitions Of Some Useful Terms:

An "object_name" is the name of a variable or the name of a constant.

An "object_descriptor" is the association between an object_name, the type of the named object and the value of the named object. Currently type of object can only be "variable" or "constant".

```
object = ( object_name object_type  object_value);
```

An "object_list" is a list of object_descriptors;

i.e.

```
object_list = ( object1  object2  object3 ... object_n);
    where each object_i is an object_descriptor;
```

A "value_list" is a list of values.

"Consume_objects" is a list of objects referenced in the consume expression of a statement. Consume_objects may contain zero or more members in its object_list.

"Produce_objects" is a list of objects referenced in the produce expression of a statement. Produce_objects may contain zero or more members in its object_list.

"Condition_objects" is a list of objects referenced in a condition expression.

"Consume_values" is a value_list containing the values of consume_objects.

"produce_values" is a value_list containing the values of produce_objects.

"condition_values" is a value_list containing the values of condition_objects of a statement.

"New_values" is a list of values obtained by executing a procedure expression. New_values may contain zero or more members in the value_list.

"Environment" of an operator is an object_list containing all objects referenced in the operator.

4.3 Working Of The EML machine: Informal Description:

4.3.1 Sequential Execution Of Statements:

An EML operator consists of a sequence of executable statements. Execution of an EML operator starts at the receive statement. The EML machine attempts to execute each successive statement in the sequence. Execution of an EML operator stops after executing its send statement. It ought to be noted that the syntax of the EML language does not permit any jump ahead or jump back during execution. In von Neumann languages, looping constructs invariably require backward branching to execute a block of statements over and over again. In the EML language the loop construct is conceived as an atomic operation. In von Neumann languages, forward jumping often is required in the "if-then-else" constructs. During execution, if the "if" condition fails, the von Neumann machine will jump over the block of statements belonging to the "if" construct. In contrast, in EML, if the condition expression associated with a statement evaluates to false, then the EML machine simply attempts to execute the next statement in the sequence. No branching ahead is permitted.

4.3.2 Executability Of A Statement:

It is important to understand when an EML statement may be executed and what happens if a statement executes or fails to execute. We first describe what governs the executability of a statement and then we shall indicate what happens to the environment in the EML machine when a statement gets executed.

It has been stated earlier that each EML statement contains a procedure expression . A statement can be executed only if its procedure expression can be executed.

When the EML machine reaches a statement, it first makes sure if the procedure expression is ready to be executed. A procedure expression is said to be ready for execution if none of the values in its list of consume_values, nor any value in its list of condition_values is empty. A simple statement will be executed if its procedure expression is ready to be executed. After executing such a statement the EML machine will proceed to examine the executability of the next statement in the sequence. If even a single object in the consume_objects of a procedure expression is associated with the value empty, then the procedure expression is not ready for execution. Such a statement will not be executed at run time. The EML machine will ignore the statement and proceed to the next statement in sequence.

When the EML machine reaches a conditional or a loop statement, it does does the following:

(a) first it examines the readiness condition of its procedure expression ;

(b) if the procedure expression is ready, then it evaluates the associated condition expression ;

(c) if the condition expression evaluates to true then the machine executes the procedure expression .

If either of (a) or (b) fails then the statement is not executed; the machine proceeds to examine the next statement in sequence.

An important difference between EML and the von

Neumann languages may be noted here. In the von Neumann languages there is no notion of readiness for execution of an assignment statement. In these languages, if at run time an assignment statement is reached then the assignment statement WILL be executed. Once reached, the executability of the assignment statement is not determined by the values of variables participating in the assignment statement. In EML, the values of consume variables and condition variables do determine the executability of a statement.

4.3.3 Effect Of Execution Of A Statement:

We now describe how the environment of an EML operator is modified by executing a statement. As defined earlier, the environment of an operator consists of the objects referenced within the operator. At run time the names are associated with values. The environment of an operator changes if the value associated with one or more names are modified. The changes happening in the environment of an operator Pi during the execution of its j_th statement may be visualized as follows. Assume the procedure expression of the j_th statement to be:

" Pj(Xj) = Xk".

In other words the operator Pi is invoking the operator Pj with variable Xj. Assume at this time the names Xj and Xk are associated with values xj and xk respectively. Obviously xj is not empty or else the procedure expression could not be executed. Execution of a procedure expression takes place in the following phases:

(1) Consumption;
(2) Bus_out;
(3) Transform or mapping;
(4) Bus_in;
(5) Production;

(1) In the consumption_phase, the value(s) associated with the consume variables are detached from their names to generate a value-list. The consume variables themselves thus become associated with empty values. In this example, xj is detached from Xj; the value list consists of only one value xj. The name Xj is associated with the empty value E. Conceptually the variable Xj is said to be consumed in the process.

(2) In this phase, the value-list generated in step 1 is bussed out to the environment of the invoked operator, where they are bound to name(s) referred in the receive statement of the invoked operator. In this example, the value xj is transported to the environment of Pj and associated with some name in the environment of Pj. This action occurs by executing the receive statement of the operator Pj.

(3) In the mapping phase, the invoked operator, Pj in this instance, is executed till it reaches its send statement. When the send statement is executed in the invoked operator a new value-list of zero or more elements may be generated for transporting back to the invoking operator.

When the send statement in Pj is executed then a new value xk' say, is generated and environment of Pj disappears. The input values xj are thus mapped to values xk'. The mapping phase is completed with the termination of execution of the invoked operator.

(4) In this phase the mapped values from phase(3) are bussed into the environment of the invoking operator. The value_list xk' is transported to Pi. Note that the value_list xk' may contain zero or more values; but none of the values may contain the special value E.

(5) The value-list obtained from the invoked operator is then associated with the names in the produce expression of the invoking operator. New values are thus said to be produced in the environment of Pi. Execution of a procedure expression is then said to be completed. In this example , the value xk' is now associated with the name Xk in the environment of Pi. The variable Xk is said to be 'produced' in the environment of Pi.

It should be noted that the environment of Pi, the operator under analysis, changes in steps (1) and (5). We will later define the functions consume update, produce update and state update to formally specify these changes in the environment. The function get value, map values will be defined to specify the process of reading value(s) associated with a name or list of names.

Two comments are worth mentioning at this point. First, in the concept described above, values are transported across environments for modification. At no time, is a value associated with more than one name. This is in sharp contrast to the von Neumann Languages which, through a variety of mechanisms, allow a value to be referenced by more than one name.

Second, the invoked operator, i.e. Pj in this instance, always operates within its own environment. Pj never modifies values associated with names outside of its own scope. There is no overlap in the name_space or value_space of Pj with any other operator. The only channel of communication between Pj and the external world are the channels through which the values are received and sent. In contrast, the von Neumann languages allow overlap of both name_space and value_space between procedures in the form of global variables and by allowing variables to be passed by reference. This is the basic argument that all EML operators may be viewed as a function from environments external to the operator itself. Thus Pi may view Pj as a function although Pj (like Pi) internally retains the notion of state and environment. Since Pj may be viewed as a function, the effect of executing Pj with some input value ought to be described as an expression. The method of deriving such an expression for an arbitrary but syntactically valid operator is the main focus of this chapter and will be described in the later sections. At this point, we will assume that the effect of executing a syntactically valid operator can be captured by an expression called the equivalent function of the operator.

4.3.4 Values generated by execution of a statement:

We have indicated above what governs the executability of a statement during execution of an operator; also we have indicated the kind of changes occurring in the environment of the operator during execution of a statement. Now we relate the values generated by execution of a statement to the values of variables referenced in the statement. Each statement has exactly one procedure expression; and each procedure expression has exactly one operator. We have argued that the operation of the invoked operator may be viewed as a function acting on the values supplied to the operator. The statements of the EML language are of either of the three following forms:

```
       Pj(Xj) = (Xk);
Or   Ci  Pj(Xj) = (Xk);
Or  loop [ Ci(Xj)  Pj(Xj) = (Xj);]
```

We will define functions simple eval, conditional eval and loop eval to specify the values generated by executing the procedure expression in a simple, conditional or loop statement.

4.4 Some Useful Functions:

We will now define some functions which will be useful to describe the working of the EML_machine. We will use notation normally employed to describe functions in functional languages (Henderson '80). The functions car, cdr, cons and the item NIL have their normal meaning. In many cases we will also describe the type of the function. To simplify the type description we use the following convention:

```
 Type of name of variable/ constant = N;
 Type of value of variable/constant = V;
 Type of name:
        T = { "variable", "constant" };
 Type of an object: O where O::(NxTxV);
 Type of list of any of the above type
        = LIST( TYPE);
 Type of an arithmetic or
        boolean expression: EX;
 Type of a statement: ST;
 Type of a function mapping values
        Dj to Dk: (Dj->Dk);
```

Function init_vars: This function is used to initialize the values of variables. The function returns a list of objects.

```
 Type of init_vars:LIST(N)->LIST(O);

 init_vars( V_NAMES)
  = if (car(V_NAMES)= NIL)
    then (NIL)
    else ( cons (list
                 ((car V_NAMES), "variable", "E"),
                  init_vars( cdr(V_NAMES)));
```

Function init_const: This function is used to initialize a list of names to values and declare that these are constants. The function returns a list of objects.

```
 Type of init_const: (LIST(N)*LIST(V))->LIST(O);

 init_const( C_NAMES, C_VALS)
  = if (car(C_NAMES) = NIL)
    then (NIL)
    else ( cons( list
           ((car(C_NAMES), "constant", car(C_VALS))),
             init_const( cdr(C_NAMES), cdr (C_VALS)));
```

Function concat: This function accepts two object_lists and merges them to one object_list.

```
 Type of concat: (LIST(O)*LIST(O))->(LIST(O);

  concat ( V_ENV, C_ENV)
   = if (V_ENV = NIL)
       then C_ENV
       else
       if( cdr(V_ENV) = NIL)
          then cons(car(V_ENV), C_ENV)
          else
          cons( car(V_ENV),
            concat(cdr(V_ENV), C_ENV));
```

Function init_env: This function is used to create a new environment before an operator is executed. The function receives a list of names of variables(V_NAMES), a list of names for constants (C_NAMES) and a list of values(C_VALS) with which to initialize the constants. The function returns an environment which in effect is a list of objects.

```
 Type of init_env: (LIST(N)*LIST(N)*LIST(V))
                         -> LIST(O);

 init_env (V_NAMES, C_NAMES, C_VALS)
  = concat( init_vars(V_NAMES),
            init_const(C_NAMES,C_VALS));
```

Function none_empty: This function examines a list of values to decide if any of the values is EMPTY. If so it returns false otherwise it returns true.

```
Type of none_empty: LIST(V)->{TRUE,FALSE};

none_empty( VALUES)
 = if( VALUES = NIL)
   then  TRUE
   else if (car(VALUES) = "EMPTY")
          then FALSE
          else  none_empty( cdr(VALUES));
```

Function consume: This function empties the value of variable object.

```
Type of consume: O->O;

consume( NAME TYPE VALUE)
 = if (TYPE = "variable")
        then <NAME, TYPE," EMPTY">
        else <NAME TYPE VALUE>;
```

Function put_value: This function associates a variable object to a new value.

```
Type of put_value: O->O;

put_value( (NAME TYPE VALUE), V1)
  if ( TYPE = "variable")
        then  <NAME, TYPE, V1>
        else  <NAME, TYPE, VALUE>;
```

Function consume_update: This function consumes the values in a list of objects.

```
Type of consume_update: LIST(O)->LIST(O);

consume_update( OB_LIST)
= if ( OB_LIST = NIL)
   then NIL
   else
   (cons  (consume car(OB_LIST))
           consume_update( cdr(OB_LIST)) );
```

Function attach_values: This function associates new values to a list of variable objects.

```
Type of attach_values:
        (LIST(O)*LIST(V)) -> LIST(O);

attach_values(OB_LIST,V_LIST)
 = if( OB_LIST = NIL)
   then NIL
   else
     cons
      ( put_value(car(OB_LIST), car(V_LIST)),
        attach_value( cdr(OB_LIST), cdr(V_LIST)));
```

Function get_value: This function maps a name (NAME) to its value in a given environment represented by OB_LIST.

```
 Type of get_value: (LIST(O)*N)->V;

 get_value (OB_LIST, NAME)
  = if(OB_LIST= NIL)
      then NIL
     else if( car(car(OB_LIST) = NAME)
            then (cdr(cdr(car(OB_LIST)))
             else get_value( cdr(OB_LIST), NAME);
```

Function map_values: This function maps a list of names (NAME_LIST) to their list of values in the environment OB_LIST;

```
 Type of map_values: (LIST(O)*LIST(N)) -> LIST(V);

 map_values(OB_LIST, NAME_LIST)
 = if (NAME_LIST = NIL)
     then NIL
     else
      cons( get_value(OB_LIST, car(NAME_LIST)),
            map_values(OB_LIST, cdr(NAME_LIST)));
```

Function get_object: This function maps a name (NAME) to its object in the environment (OB_LIST).

```
Type get_object: (LIST(O)xN) -> O;

get_object( OB_LIST, NAME)
= if (OB_LIST = NIL)
   then NIL
   else if( car(car(OB_LIST) = NAME)
          then (car(OB_LIST))
          else get_object( cdr(OB_LIST), NAME);
```

Function map_objects: This function maps a list of names(NAME_LIST) to a list of objects in the environment OB_LIST.

```
Type map_objects: (LIST(O)xLIST(N)) -> LIST(O);

map_objects(OB_LIST, NAME_LIST)
 = if( NAME_LIST = NIL)
    then (NIL)
    else
      cons ( get_object(OB_LIST, car(NAME_LIST)),
           map_objects(OB_LIST, cdr(NAME_LIST)) );
```

Function delete_object: This function deletes an object (OBJECT) from an environment (OB_LIST).

```
Type delete_object: (LIST(O) x O) -> LIST(O);

delete_object (OB_LIST, OBJECT)
 = if (OB_LIST = NIL)
   then (NIL)
   else if (get_name( car(OB_LIST) = get_name(OBJECT))
          then (cdr(OB_LIST))
          else  cons( car(OB_LIST),
             delete_object( cdr(OB_LIST), OBJECT));
```

Function remove_duplicates: This function deletes from the environment OB_LIST1, the objects which are also present in the environment OB_LIST2.

```
 Type remove_duplicates: (LIST(O) x LIST(O))
                           -> LIST(O);

 remove_duplicates ( OB_LIST1, OB_LIST2)
  = if ( OB_LIST2 = NIL)
      then NIL
      else remove_duplicates
             (delete_object( OB_LIST1, car(OB_LIST2)),
                cdr(OB_LIST2));
```

Function state_update: This function is used to update the environment of an operator when a procedure expression is executed by the EML_machine.

```
  state_update( OB_LIST1, OB_LIST2)
   = concat ( remove_duplicates( OB_LIST1,OB_LIST2),
                OB_LIST2);

  Type state_update: (LIST(O)*LIST(O)) -> LIST(O);
```

Function member: This function determines if an item is present in a list of symbols (LIST).

```
  Type of member: (N x LIST(N)) -> {TRUE,FALSE};

  member ( ITEM, LIST)
  = if( car(LIST) = NIL)
        then FALSE
        else if (car(LIST) = ITEM)
              then TRUE
              else ( member( ITEM, cdr(LIST)));
```

Function bool_eval: This function evaluates a boolean expression in an environment.

```
 Type of bool_eval: (EX x LIST(O)) -> {TRUE,FALSE};

bool_eval( OP, EXP, ENV)
 = if (OP= gt)
   then (gt (map_eval car(EXP) ENV)
             (map_eval cdr(EXP) ENV))
   else
    if (OP= lt)
      then (lt (map_eval car(EXP) ENV)
             (map_eval cdr(EXP) ENV))
   else
    if (OP= le)
      then (le (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= ge)
      then (ge (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= eq)
     then (eq (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= ne)
     then(ne (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= and)
     then (and (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= or)
     then (or (map_eval car(EXP) ENV)
               (map_eval cdr(EXP) ENV))
   else
    if (OP= not)
     then not( map_eval( car(EXP), ENV);
```

Function eval: This function evaluates an arithmetic expression in a given environment.

```
Type of eval: (EX x LIST(O)) -> V;
eval( OP, EXP, ENV)
= if ( OP = add)
   then (add (map_eval car(EXP) ENV)
                (map_eval cdr(EXP) ENV))
  else
  if ( OP = sub)
   then (sub (map_eval car(EXP) ENV)
                (map_eval cdr(EXP) ENV))
  else
  if ( OP = mult)
   then (mult (map_eval car(EXP) ENV)
                (map_eval cdr(EXP) ENV))
  else
  if ( OP = div)
   then (div (map_eval car(EXP) ENV)
                (map_eval cdr(EXP) ENV))
  else
  if ( OP = mod)
   then (mod (map_eval car(EXP) ENV)
                (map_eval cdr(EXP) ENV));
```

Function map_eval: This function evaluates an expression in an environment.

```
Type map_eval: (EX x LIST(O)) -> V;

map_eval ( EXP, ENV)
= if ( car(EXP) = NIL)
   then NIL
   else if (member( car(EXP),
              <gt, ge, lt, le, eq, ne, and, or,  not>))
         then ( bool_eval( car(EXP), cdr(EXP), ENV))
         else  if (member( car(EXP),
                  <add, sub, mult, div, mod>))
                then ( eval( car(EXP), cdr(EXP), ENV))
                 else   get_value( ENV, car(EXP));
```

In the following we describe some functions which specify the effects of executing an EML statement. We assume that:

```
ENV: represents the environment in which the EML
      machine attempts to execute the statement;
VCONS,VCOND,VPROD :
      represents names of objects in the
      consume_expression, condition_expression,
      produce_expression of the statement;
BOOL_EX: represents the condition_expression
      of the statement;
FUN: represents the equivalent_function of the
      operator in the procedure expression
      of the statement.
```

Function simple_eval: This function expresses the new values available to an environment when a procedure expression is executed.

```
Type simple_eval: (LIST(O) x LIST(N) x (D_j->D_k))
                         -> LIST(V);
simple_eval( ENV,VCONS, FUN)
  = FUN(map_values(ENV,VCONS);
```

Function cond_eval: This function describes the new values obtained by executing a conditional statement.

```
Type of cond_eval:
  (LIST(O) x EX x LIST(N) x (D_j->D_k) x LIST(N))
        -> LIST(V);

cond_eval( ENV,BOOL_EXP,VCONS,FUN,VPROD)
 = if ( bool_eval
       ( car(BOOL_EXP), cdr(BOOL_EXP), ENV) = TRUE)
    then ( simple_eval( ENV,VCONS,FUN))
       else (map_values(ENV,VPROD));
```

If the boolean expression (BOOL_EXP) evaluates to true in the environment ENV, then the function executes simple_eval and returns the values; otherwise cond_eval returns the values associated with the names in VPROD in the environment ENV, i.e the procedure expression is not executed.

Function simple_update: This function expresses change on the environment when the EML_machine attempts to execute a simple statement.

```
Type of simple_update:
  (LIST(O) x LIST(N) x (D_j->D_k) x LIST(N))
      -> LIST(O);
simple_update( ENV,VCONS,FUN,VPROD)
 = if (none_empty(map_values(ENV,VCONS)) = FALSE)
    then (ENV)
   else
    (state_update( ENV,
      state_update
       ( consume_update(map_objects(ENV,VCONS)),
          attach_values( map_objects(ENV,VPROD),
              simple_eval( ENV,VCONS,FUN))));
```

Function conditional_update: This function expresses the effect on the the environment when the EML machine attempts to execute a conditional statement.

```
Type of conditional_update:
  (LIST(O) x EX x LIST(N) x LIST(N)
      x (D_j->D_k) x LIST(N))
        -> LIST(O);

conditional_update(ENV,BOOL_EXP,
              VCOND,VCONS,FUN,VPROD)
 = if (and
        (and( none_empty(map_values(ENV,VCOND)),
              none_empty(map_values(ENV,VCONS)))),
         boole_eval( car(BOOL_EXP),
                       cdr(BOOL_EXP), ENV)
       = FALSE)
    then (ENV)
    else
    ( state_update( ENV,
        state_update(
         consume_update(map_objects(ENV,VCONS),
          attach_values( map_objects(ENV,VPROD),
           cond_eval(ENV, BOOL_EXP,VCONS,FUN,VPROD)))));
```

Function loop_eval: This function describes the new values obtained by executing a loop_statement.

```
 Type of loop_eval:
   (LIST(O) x EX x LIST(N) x (D_j->D_K) x LIST(N))
        -> LIST(V);

 loop_eval( ENV, BOOL_EXP, VCONS, FUN)
  = if (bool_eval( car(BOOL_EXP), cdr(BOOL_EXP), ENV)
        = FALSE)
        then (map_values( ENV,VCONS))
     else
       loop_eval( simple_update(ENV,VCONS,FUN,VCONS),
                  BOOL_EXP, VCONS, FUN)
```

If the boolean expression BOOL_EXP evaluates to false in environment ENV, then loop_eval returns the values of names VCONS in ENV, i.e. looping stops; otherwise it recursively executes the procedure expression and updating the environment. The values returned by the loop_eval function may be obtained by the following inductive reasoning. Let ENVj(0) and c_j(0) represent the environment ENV and the values associated with the names VCONS at the beginning. If the BOOL_EXP evaluates to false, then the function returns the values associated with VCONS in ENVj(0), i.e. it returns c_j(0). Otherwise, it updates the environment to ENVj(1) and goes through next iteration. Let c_j(1) represent the values of VCONS in ENVj(1). Since

ENVj(1) = simple_update(ENVj(0), VCONS, FUN,VCONS)

we conclude that only the values of VCONS alters as the environment is updated. Since simple_update utilizes 'simple_eval(ENVj(0),VCONS,FUN)' to obtain new values, we find that: c_j(1) = FUN(c_j(0)). As long as iteration continues, only the values of VCONS changes. They can be related to the initial values as follows:

```
   c_j(0) = c_j           in ENVj(0) = ENVj;
   c_j(1) = FUN(c_j)      in ENVj(1);
      ....
   c_j(k) = FUN(c_j(k-1))    in  ENVj(k);

   Hence: cj_(k) = FUN(FUN(FUN...FUN(c_j)..))_k;
```

Also the recursion terminates when the BOOL_EXP evaluates to false. Let us assume that the recursion terminates after k_iterations. Hence the BOOL_EXP must evaluate to true during 0 to (k-1) iterations and must evaluate to false after k_iterations. Hence k is such a positive integer that the two conditions below are satisfied.

```
 bool_eval( car(BOOL_EXP), cdr(BOOL_EXP), ENVj(i))
       = TRUE  for 0<=i < k;
and
 bool_eval( car(BOOL_EXP), cdr(BOOL_EXP), ENVj(i))
       = FALSE for  i=k;
```

Thus given the arguments of the function loop_eval it is possible to reason about k and also it is possible to relate c_j(k) to c_j i.e it is possible to relate the values associated with VCONS before and after the execution of the loop_statement.

Function loop_update: This function expresses the effect on the environment when the EML machine attempts to execute a loop_statement in a given environment.

```
Type of loop_update:
(LIST(O) x EX x LIST(N) x LIST(N)
       x (D_j->D_k) x LIST(N))
          -> LIST(O);

loop_update(ENV,BOOL_EXP,VCOND,VCONS,FUN)
  = if (and
          (and
            (none_empty(map_values(ENV,VCOND)),
              none_empty(map_values(ENV,VCONS))
            ),
            bool_eval( car(BOOL_EXP),
                 cdr(BOOL_EXP), ENV)
          ) = FALSE)
          then (ENV)
       else
       ( state_update( ENV,
          state_update(
           consume_update( map_objects(ENV,VCONS)),
            attach_values( map_objects(ENV,VCONS),
               loop_eval(ENV,BOOL_EXP,VCONS,FUN)))));
```

Function receive_update: This function specifies the effect of executing a receive statement in an environment ENV. In the following ARGS represent the argument_list,i.e. a list of values with which an operator is invoked.

```
Type of receive_update:
  (LIST(O) x LIST(V) x LIST(N)) -> LIST(O);

receive_update ( ENV, ARGS, VPROD)
 = if (none_empty(ARGS))
          then state_update( ENV,
             attach_values
                 (map_objects(ENV,VPROD),ARGS))
        else (error);
```

Function send_update: This function represents the effect of executing the send statement of an operator.

```
Type of send_update:
   (LIST(O) x LIST(N)) -> LIST(V);

send_update( ENV, VCONS)
  = if( none_empty(map_values(ENV,VCONS)))
      then (map_values(ENV,VCONS))
      else (error);
```

Function env_update: This function describes the effect of executing a statement in the body of the operator. Depending on the type of the statement, env_update calls other functions and finally returns an updated environment.

```
Type of env_update:
  ( LIST(O) x
      (ST x EX x LIST(N) x LIST(N)
        x (D_j->D_k) x LIST(N))
          -> LIST(O);

env_update ( ENV,
      <TYPE, BOOL_EXP, VCOND, VCONS, FUN, VPROD> )
  = if ( TYPE= simple) then
       simple_update(ENV,VCONS,FUN,VPROD)
     else
      if (TYPE= conditional) then
        conditional_update( ENV, BOOL_EXP,
              VCOND,VCONS,FUN,VPROD)
    else
      if (TYPE=loop) then
        loop_update(ENV,BOOL_EXP,VCOND,VCONS,FUN)
    else
      (error)
```

4.5 Semantic specification of the EML language:

We will first provide some formal definitions, then provide the abstract syntax of the language and finally give equations to specify the effect of executing EML statements and EML-operators.

4.5.1 Formal Definitions:

We assume the existence of:

```
V: a set of  names of objects
       (variables and constants);
D: a set of data  values;
E: a distinguished element of D,
       representing the value empty.
B: a set of primitive boolean functions.
  Bi,j: D_j ->{T,F}  of type <j,boolean>,
  where T,F are distinguished elements of D.
O: a symbol neither in D or V;
P: a set of primitive operators
    Pi,j,k: D_j -> D_k, of type <j,k>;
F: a set of function names;
TYPE: a set of three distinguished symbols
       { "simple", "conditional","loop"};
NT: a set of two distinguished symbols
       { "variable", "constant"};
```

A function type is either: <i,boolean> or <i,j > where i and j are integers. This means two types of functions are recognized in EML. The first type is of the type boolean; it accepts a sequence of i values and maps the sequence to a boolean value. The second type accepts accepts a sequence of i values and maps it to another sequence of j values where j >= 0.

A function_environment FE is a mapping:

```
FE: F->T x O
where: T is a function type;
    and   O: is a primitive operator
             in P or an EML operator
                to be defined below.
```

This means we assume the existence of a function FE which takes the name of a function F and returns the operator for the function and the type of the operator.

The operator may be a language defined operator in P or a user defined operator.

We will refer to T as Type(F,FE) and to an operator O as Operator(F,FE). We define

```
Type(FUN,FE) = T;
DEF (FUN, FE) = O;
```

VARS is defined to be the tuple: VARS= <V_NAMES, C_NAMES, C_VALS> where: V_NAMES is in V, C_NAME is in V, and C_VALS is in D. V_NAME and C_NAME represent the lists of names of variables and constants used in the operator. C_VALS is a list of values which are to be bound to the names in C_NAMES.

4.5.2 Abstract syntax:

Boolean Expression BE:

Given a functional environment FE, a boolean expression over a set of variables is defined as follows:

BE ::=<FUN,A1,A2..A_i>
is a boolean expression over V if for each A_j 1<=j<=i A_j is in D or A_j is in VARS, or A_j is itself a boolean expression over VARS; and if Type(FUN,FE) = < k,boolean>.

EML Statement ST:

Given a functional environment FE, an EML statement is defined over a set of object_names VARS as follows: The tuple

```
ST ::= <S_TYPE, BE, VCOND, VCONS, FUN, VPROD>
is an EML statement if:
(1) VCOND is in VARS; these are names referred in the
        condition_expression of the statement.
(2) BE is a boolean expression over VCOND;
(3) VCONS is in VARS; these are names referred in the
        consume expression of the statement.
(4) VPROD is in VARS; these are names referred in the
        produce_expression of the statement.
(5) FUN is in F; this is the function expression of the
        operator named in the procedure_expression of
        statement.
(6) Type ( FUN,FE) = <j,k>;
(7) S_TYPE is in TYPE;
```

The receive and statements of the EML language are special cases of the above and are defined as follows. These statements are defined by the tuples R and S as follows:

```
 R ::=< VPROD >;
 S ::= <VCONS>;
```

where VCONS, VPROD are in VARS. These are the objects referred in the produce expression and consume expressions of the receive and send statements of the operator.

EML operator:

Given a function environment FE, an EML operator O of type <j,k> is defined by the tuple:

```
Operator ::= <VARS, R, BODY, S>
  where:
  (1) VARS is a tuple <V_NAMES, C_NAMES, C_VALS>
                defined earlier.
  (2) BODY is a sequence <ST1,ST2,...ST_k>
        where ST_i is an EML statement over VARS.
  (3) R and S are tuples of receive and send statements
      defined over VARS;
```

Now we can formally define an EML program. An EML program is a tuple:

Program ::= <V,D,F,TRUE, FALSE, B, P, O, FE, M>

of type <i,j> if:

(1) V,D,F are all disjoint sets of symbols;
(2) TRUE,FALSE are distinguished elements of D;
(3) B is a set of primitive boolean operators;
(4) P is a set of primitive operators;
(5) O is a set of EML operators;
(6) FE: is a function environment;
(7) M is an EML operator, the main operator.

4.5.3 Effect of executing EML operator:

We can now begin to define the effect of executing an EML program or operator. Let O an EML operator of the type <I,J> be described as follows:

```
 O = < VARS, R, BODY, S>
   = < <V_NAMES, C_NAMES, C_VALS>,
        R,
        <ST1,ST2,..ST_k>,
        S>
 where each of ST_i is of the form:
  ST_i = <TYPE_i,VCOND_i, BE_i,CO_i, FUN_i, PO_i>;
```

We define the environments assumed by the operator O during execution to be as follows:

```
  ENV(O,ARGS)(0) = init_env( V_NAMES, C_NAMES, C_VALS);

  ENV(O,ARGS)(1)
        = receive_update ( ENV(O,ARGS)(0), R);


  For 1<i<=k,
  ENV(O,ARGS)(i+1)
        = env_update( ENV(O,ARGS)(i), ST_i);
        = env_update( ENV(O,ARGS)(i),
             <TYPE_i,VCOND_i,BE_i,CO_i,FUN_i,PO_i> );
```

The values returned by the operator after termination of execution is defined by:

```
 FINAL_VALUE(O,ARGS)
       =if (member (O, P))
        then O(ARGS)
        else send_update( ENV(O,ARGS)(k+1), S);
```

Note that the send function returns a list of values. Hence the final effect of executing the operator is to map the arguments ARGS to a list of values. We define a second order function EQ_FUNC to obtain the equivalent function of an operator. EQ_FUNC is defined as follows:

```
  EQ_FUNC(O) (LAMBDA(ARGS)(FINAL_VALUE(O,ARGS)));
```

Hence the equivalent function of an operator O is obtained by applying the function EQ_FUNC on the tuple representing O. The equivalent function of the operator O ,i.e. EQ_FUNC(O) is such a function that it maps the argument list ARGS to the value_list FINAL_VALUE(O,ARGS). In other words, EQ_FUNC is a mapping from the textual description of an operator to a function expression representing the effect of executing the operator.

4.6 Meaning Of Executing An Operator:

We can now use the above formalism to analyze the meaning of executing an operator. This can be done in three stages:

(1) Analyze the text description of the operator to generate its abstract form. That is generate the tuples representing the abstract form described in previous section. We assume that there is a parsing function

"abstract" available to achieve this (see below).

(2) Use the abstract form and an argument list to generate successive environments of the operator employing the specifications and functions described in sections 4.5 and 4.4. The successive environments provide the internal view of the operator during execution.

(3) Obtain the expression 'FINAL_VALUE' as described in section 4.5 to obtain the values returned by the operator after termination of execution. The values FINAL_VALUE(O,ARG) represent the external view of the operator.

We assume the existence of the function abstract defined as follows:

```
abstract
  = LAMBDA( o.operator_name)
      if ( o.operator_name is in P)
       then (operator_name)
       else
        (< <V_NAMES,C_NAMES,C_VAL>, R, BODY, S>);
         where the meaning of the tuple is the same
              as in section 4.5.
```

If the operator_name is one of the primitive operators (P), i.e. if it is one of the operators defined in the EML language, then the operator name is returned. Otherwise the text is processed to obtain the necessary tuple. In effect a parser of the language modified by a simple rule could implement the function "abstract". If a semantic entity is not applicable to a syntactic construct, then the corresponding semantic entity could be replaced by 'NIL'; e.g. if the statement ST_i is a simple statement then VCOND_i = NIL, and BOOL_EXP_i = NIL as well. We will not describe the mapping abstract any farther in this paper. The result of its application on several operators will be shown below.

The application of semantic specification on the tuple O produce successive environments ENV(O)(i) for 0<= i <= (k+1) if the body of the operator contains k statements. The receive statement is executed in ENV(O)(0) and the send statement is executed in ENV(O)(k+1). The attempted execution of the i_th statement in the body produces ENV(O)(i+1). We will present the result of application of the semantic specification in the form of a table. The i_th row of the table will contain the objects in ENV(O)(i). We will see that each value in the table is either a constant or a value present in the argument list or an expression involving only constants or members of the argument list. If the expression becomes too cluttered, we will replaced it with a symbolic name; the symbolic name will be will be defined below the table.

We will adopt the following convention in naming the equivalent_function of a user_defined operator. The name of the equivalent_function of an operator will be obtained by deleting the string "o." from the name of the operator. Thus equivalent_function of the operator o.new could be written as:

```
EQ_FUNC( o.new)
    = new
    = LAMBDA(ARGS) FINAL_VALUE( abstract(o.new), ARGS);
```

We will illustrate the method with three operators: o.sum_inc, o.integer_sum and o.difference. The operator o.sum_inc uses only simple statements. The operator o.integer_sum contains a loop statement and also invokes the operator o.sum_inc. The operator o.difference uses conditional as well as simple statements.

The operator o.sum_inc is shown in figure 4.1; its abstract form is in figure 4.2. The environments of the operator during execution is shown in table 4.1.

Figure 4.1
Operator o.sum_inc

```
o.sum_inc
{
  receive () =  (v.num, v.sum);
  copy ( v.num, 2) = (v.num, v.num1);
  add (v.num, v.sum) = (v.sum);
  add (v.num1, 1) =  (v.num);
  send (v.num, v.sum).
}
```

```
              Figure  4.2
      Abstract form of operator o.sum_inc

    abstract(o.sum_inc)
      = < <V_NAME, C_NAME, C_VAL>, R, BODY, S>
     where:
     {  V_NAME = <v.S, v.T, v.S1>;
        C_NAME = <c.TWO, c.ONE>;
        C_VAL  = < 2, 1>;
        R  = <v.S, v.T>;
        BODY = <ST_1,ST_2,ST_3>
          where:
          { ST_1 = <simple, NIL, NIL, <v.S,c.TWO>,
                         copy, <v.S,v.S1>>;
            ST_2 = <simple, NIL,NIL, <v.S, v.T>,
                         add, <v.S>>;
            ST_3 = <simple, NIL, NIL, <v.S1,1>,
                         add, <v.S>>;
          };
         S = <v.S, v.T>;
      };
```

It is obvious from the table 4.1, that the final values returned by the executing the operator o.sum_inc is:

```
  FINAL_VALUE( (abstract(o.sum_inc), <num, sum>
        = send_update ( ENV(4), (v.num,v.sum));
        = < (num+1), (num+sum) >;
```

Hence the equivalent_function of the operator o.sum_inc could be written as:

```
  sum_inc = LAMBDA(num,sum)
          ( (num+1), (num+sum));
```

Table 4.1
Environments of o.sum_inc (num,sum)

Object_ names	Object_values				
	ENV(0)	ENV(1)	ENV(2)	ENV(3)	ENV(4)
v.S	E	num	num	num	(num+1)
v.T	E	sum	sum	(num+sum)	(num+sum)
v.S1	E	E	E	num	E
c.two	2	2	2	2	2
c.one	1	1	1	1	1

We may now analyze the execution of the operator o.integer_sum. This operator and its abstract form are described in figure 4.3 and figure 4.4.

Figure 4.3
Operator o.integer_sum

```
o.integer_sum
 { receive () = (v.last);
                           /* Statement 0 */
   copy (0)   = (v.total, v.start);
                           /* Statement 1 */
   loop
   [ if (le ( v.start, v.last))
       o.sum_inc ( v.start, v.total)
             = (v.start, v.total);]
                           /* Statement 2 */
   send ( v.total).        /* Statement 3 */
 }
```

Figure 4.4
Abstract form of the operator o.integer_sum

```
Abstract( o.integer_sum)
   =< VARS, R, <BODY>, S>
   where:
   { VARS = < (v.last, v.start, v.total) ,
            (c.zero c.two), (0 ,2) >;
     R =<v.last>;
     BODY = < ST1,ST2>
     where:
     { ST1 = <simple, NIL, NIL, <c.zero,c.two>,
              copy, <v.total,v.start>>;
       ST2 = <loop, <le<v.start,v.last>>,
              <v.start, v.last>,
               EQ_FUNC( o.sum_inc),
              (v.start,v.last));
     };
     S = (v.total);
   };
```

In order to find the final values obtained after executing the the operator, we must evaluate ENV(3) as shown in table 4.2. The loop_update function in turn invokes the recursive function loop_eval. Hence we need to evaluate the loop_eval function shown below. The environments assumed by the operator o.integer_sum during the execution of the loop_statement are shown in table 4.2A. As stated earlier, we will use inductive reasoning to evaluate the function.

```
 loop_eval (ENV(2), (le(v.start, v.last)),
                  (v.start,v.total), sum_inc);
```

Let (S0,T0) and (S_k, T_k) be the values associated with the names (v.start,v.total) before and after termination of the loop_eval function. (S0,T0) are the values associated with (v.start, v.last) in ENV(2). These values are easily obtained as follows:

```
   S0 = get_value( ENV(2), v.start) = 0;
   T0 = get_value( ENV(2), v.total) = 0;
```

By applying induction, the values assumed by the variables (v.start,v.total) at successive iterations are obtained as follows:

```
  (S0, T0) = (0,0);
  (S1, T1) = num_inc(S0,T0) = (1,0);
  (S2, T2) = num_inc(S1,T1) = (2, (0+1));
  (S3, T3) = num_inc(S2,T2) = (3, (0+1+2));
        ....
  (S_k, T_k) = (k, (0+1+2+...(k-1))
             = (k, ( (k-1)*(k-2))/2);
```

If the function loop_eval terminates after k iterations then the following conditions must be true:

```
 ( bool_eval( le, v.start, v.last, ENV(2)(k-1)) = TRUE;)
 and
 ( bool_eval( le, v.start, v.last, ENV(2)(k)) = FALSE;)

 Hence:
      le ( (k-1),  num) = TRUE;
  AND le ( k, num) = FALSE;

 Hence: k = num + 1;
```

The values returned by the loop_eval are thus:

```
 loop_eval( ENV(2), (le(v.start, v.last)),
            (v.start,v.total), sum_inc)
        = (k, ( (k-1)*(k-2))/2)
        = ( (num + 1), (num *(num - 1))/2);
```

Using these values we can evaluate the loop_update function to obtain ENV(3). Also we can show that final_values obtained by executing o.integer_sum is as follows:

```
  FINAL_VALUE( ENV(4), ( num)) = num*(num-1)/2;
  Hence:
   integer_sum = LAMBDA( num)( num * (num - 1)/2);
```

It can easily be shown that if (num < 0) then the execution of the operator will generate an error.

Table 4.2
Environments of o.integer_sum (num)

Object_ names	Object_values			
	ENV(0)	ENV(1)	ENV(2)	ENV(3)
v.last	E	num	num	num
v.total	E	E	0	num*(num-1)/2
v.start	E	E	0	num
c.zero	0	0	0	0
c.two	2	2	2	2

Table 4.2A
Environments of o.integer_sum (num) during execution of the loop_statement.

Object_ names	Object_values					
	ENV(2) (0)	ENV(2) (1)	ENV(2) (2)	ENV(2) (3)	...	ENV(2) (k)
v.last	num	num	num	num		num
v.total	0	0	(0+1)	(0+1+2)	..	(0+1+2.. (k-1))
v.start	0	1	2	3	...	k
c.zero	0	0	0	0	....	0
c.two	2	2	2	2	....	2

The operator o.difference is supposed to accept two values (x,y) say. Depending on the relationship between x and y, the operator is to return a value z. The value of z should be such that:

```
   if (x > y) then z= (x-y);
   if (x < y) then z= (y-x);
      else         z = (x*x);
```

The code and abstract form of the operator o.difference are in figure 4.5 and figure 4.6. The environments are in table 4.3. It can be easily shown that :

```
  if ( x > y) then
   ENV(2) = ENV(3) = ENV(4)
        = ( (v.X E)(v.Y E)(v.Z (x-y)));
```

Thus it can be shown that:

```
  FINAL_VALUE( o.difference, (x,y))
        = send_update
           ( ENV(o.difference, (x,y))(4), (v.Z))
        = (x-y);
```

Similarly it can be shown very easily that:

```
  if ( x <y ) then
        send_update
        (ENV(o.difference, (x,y))(4), (v.Z))
        = (y-x);
  if ( x = y) then
        send (ENV(o.difference, (x,y))(4), (v.Z))
        = (x*x);
```

Figure 4.5
Operator o.difference

```
 o.difference
  { receive () = (v.X  v.Y);       /* Statement 1*/
    if ( gt (v.X,v.Y))  sub (v.X, v.Y) = v.Z;
                                   /* Statement 2*/
    if ( lt (v.X, v.Y)) sub (v.Y, v.X) = v.Z;
                                   /* Statement 3*/
    sqr (v.X)      = v.Z;          /* Statement 4*/
    send ( v.Z).
   }
```

It is obvious that if either of the conditional statements execute, then value of v.X will become <u>empty</u>, and therefore the statement 4 will not execute.

Figure 4.6
Abstract form of operator o.difference

```
abstract (o.difference)
  = < <V_NAMES, C_NAMES, C_VALS>,
      R, BODY, S>
    where:
    { V_NAMES = (v.X,v.Y,v.Z);
      C_NAMES = (NIL);
      C_VALS  = (NIL);
       R̄ = (v.X,v.Y);
       BODY = <ST1,ST2,ST3>
       where:
       { ST1 = < conditional, gt(v.X,v.Y),
                     (v.X,v.Y), (v.X,v.Y), sub,
                         (v.Z)>;
         ST2 = < conditional, (lt(v.X,v.Y)),
                     (v.X,v.Y), (v.Y,v.X), sub,
                         (v.Z)>;
         ST3 = < simple, NIL, NIL, (v.X),
                         sqr, (v.Z)>;
       };
       S = <v.Z>;
    };
```

Table 4.3
Environments of o.difference (x,y)

Object_ names	Object_values				
	ENV(0)	ENV(1)	ENV(2)	ENV(3)	ENV(4)
v.X	E	x	x2	x3	x4
v.Y	E	y	y2	y3	y3
v.Z	E	E	z2	z3	z4

```
x2 = if (x > y) then (E) else (x);
y2 = if (x > y) then (E) else (y);
z2 = if (x > y) then (x-y) else (E);
x3 = if (x < y) then (E) else (x);
y3 = if (x < y) then (E) else (y);
z3 = if (x < y) then (y-x) else (E);
x4 = if (none_empty(x3)) then (E) else (x3);
z4 = if (none_empty(x3)) then (x*x) else (z3);
```

CHAPTER 5

VERIFICATION OF EML PROGRAMS

5.1 From Program To Expressions:

The aim of program verification is to determine if the program under consideration implements the intentions of its users. It is assumed that users intentions will be supplied in the form of assertions which are to hold true at specified stages of computations of the program in question. An assertion is a relationship between the values of variables and constants. In verifying EML programs we propose to proceed in two distinct stages:

(1) Applying the semantic specification of the language, we will transform the source code of the EML operator to a set of expressions, one for each value relevant to the computation. The terms of the expression will consist of the input values (if any) to the operator, the constants known to the language and either built in operators of the language and/or user defined operators referenced within operator being verified. At any given stage of computation we hope to have a symbolic expression for each variable referenced in the operator being verified.

(2) Examine the expressions to determine if the user's assertions are satisfied by the expressions.

If the user supplied assertions are satisfied by the expression, then an operator will be said to be verified.

The above two stages in the verification process may be conceptualized as follows:

```
                        Transform
      SOURCE-CODE  -------------------> EXPRESSIONS;

      EXPRESSIONS : satisfy user supplied assertions?
```

In comparing the ease of verification of programs in EML and other imperative languages we claim the following: Given any syntactically valid EML operator, systematic application of the semantic specification of the language may generate a set of symbolic expressions. The operators in these expression are either defined in the language or are user defined operators referenced in the operator under analysis or are the meta operators <u>loop update</u> etc. described in the semantic specification of the language (see chapter 4). Similar analysis of programs in imperative languages may or may NOT be possible. In other words, there is no guarantee that we may transform every syntactically valid procedure in 'C' or PASCAL to a set of expressions which employ only language defined operators and/or user defined operators referred in the procedure. In the context of imperative languages the process to derive these expressions may be quite difficult or impossible. Often it may be necessary to be inventive to derive such expressions. We shall present examples to support this claim.

We do NOT claim that every syntactically valid EML program can be proved to be correct. Although every EML program may be easily transformed to expressions, there is no assurance that each of these programs will terminate during execution. In fact, if an EML program contains loop statements or recursion, then it will be necessary to employ inductive reasoning to determine if the loop or recursion will indeed terminate. We think, however, it may be easier to employ inductive reasoning on the algebraic expressions obtained by transforming EML programs. Similar reasoning on the source code of imperative languages may not be as easy since, because of side effects, it may be quite difficult to decide the nature of the function computed by an arbitrary loop statement.

It ought to be noted that the complexity of the second step of the verification process, i.e. the process of determining if the expressions satisfy the users assertions is dependent on the complexity of the expressions themselves. The complexity of the expressions are directly related to the complexity of the procedure. If EML does offer any advantage over the imperative languages in program verification, it lies in the ease of deriving these expressions and not in analyzing the properties of such expressions.

5.2 Verification Of An EML Operator:

We will now describe the verification process stated above with four examples. In the first example, we will analyze the properties of an EML program designed to do integer division. The analysis and verification of this program is quite straightforward. Hoare (Hoare '69) has discussed the verification of a similar program within the context of imperative languages. The second example deals with a program determining the greatest common divisor using the Eucleadian algorithm (Djikstra '76). We will show how difficult it is to derive the properties of a similar program in the imperative languages. The third example illustrates the analysis of a recursively defined operator. In the fourth example, we present the analysis of programs containing non_terminating loops.

5.2.1 Example 1:

In the first example we will discuss the verification process of an EML program intended to determine the quotient of two nonnegative integers. The program is implemented by the operator o.quotient, which in turn uses another user defined operator o.sub_and_add. The operator o.quotient is described in figure 5.1. The environments of this operator are shown in table 5.1. We will verify if the operator o.quotient implements the intention of its users. Formally speaking, this operator claims to find the quotient 'q' and the remainder 'r' obtained on dividing 'x' by 'y'. All variables are assumed to range over positive integers. The operator will be considered verified if we can show that on termination the operator satisfies the three assertions: (a) $x = r + y*q$; (b) $r < y$; and (c) $r > 0$. We farther assume that: $x > y$; and $x > 0$; and $y > 0$.

Figure 5.1
Operator o.quotient

```
o.quotient
{  Vars : v.number, v.divisor, v.quotient;

   receive() =    v.number, v.divisor;
   copy (0,1)=    v.quotient;
   loop
   [ if ( ge ( v.number, v.divisor))
      o.sub_and_add ( v.number,v.divisor,v.quotient)
             =  (v.number,v.divisor,v.quotient);]
         send ( v.quotient, v.number).
        /* number = remainder */
 }
```

Table 5.1

Environments of o.quotient(x,y)

objects	Environments			
	ENV(0)	ENV(1)	ENV(2)	ENV(3)
v.number	E	x	y	x_k
v.divisor	E	y	y	y_k
v.quotient	E	E	0	z_k
c.zero	0	0	0	0
c.one	1	1	1	1

```
(x_k, y_k, z_k)
    = loop_eval ( ENV(2),
           (ge (v.number, v.divisor),
              (v.number,v.divisor,v.quotient),
                 sub_and_add);
```

The operator o.quotient invokes the operator o.sub_and_add. Hence we analyze the latter first before deriving the effect of executing the operator o.quotient. The operator o.sub_and_add is described in the figure 5.2. Its environments are shown in table 5.2. The effect of executing the operator o.sub_and_add is given by the expression:

```
sub_and_add( x,y,q)
    = send_update( ENV(4),
          (v.number,v.divisor,v.quotient))
       where: ENV(4) is obtained from Table 5.2.
    = ( (x-y), y, (q+1));
```

Figure 5.2

Operator o.sub_and_add

```
o.sub_and_add
{ vars: v.from,v.num,v.to, v.num1;

  receive ()  =   (v.from, v.num, v.to);
  copy (v.num,2)  =  v.num, v.num1;
  sub ( v.from,v.num1) =   v.from;
  add ( v.to,1)  =    v.to;
  send ( v.from, v.to, v.num).
}
```

Table 5.2

Environments of o.sub_and_add(x, y,q)

objects	Environments				
	Env(0)	Env(1)	Env(2)	Env(3)	Env(4)
v.from	E	x	x	(x-y)	(x-y)
v.num	E	y	y	y	y
v.add_to	E	q	q	q	(q+1)
v.num1	E	E	y	E	E
c.two	2	2	2	2	2
c.one	1	1	1	1	1

Next we determine the effect of executing the operator o.quotient with input arguments (x,y). From table 5.1:

```
quotient(x,y)
     =send_update( ENV(3), (v.quotient, v.divisor))
     {where:
      ENV(3) = loop_update( ENV(2),
                  (ge(v.number, v.divisor)),
                     (v.number, v.divisor,v.quotient),
                       sub_and_add);
     };
```

In order to evaluate the loop_update expression above, we will have to evaluate the following expression (see definition of loop_update in chapter 4):

```
loop_eval ( ENV(2),
     (ge(v.number, v.divisor)),
       (v.number, v.divisor, v.quotient),
         sub_and_add);
```

The loop_eval function is evaluated by induction as indicated in chapter 4. Assume x_i,y_i,q_i are values of v.number, v.divisor, v.quotient respectively at i_th iteration of loop_eval. Hence:

```
   (x0,y0,q0) = (x,y,0);
   (x1,y1,q1) = sub_and_add (x0,y0,q0)
              = ( (x-y), y, (q+1));
```

By solving the appropriate recurrence relations we obtain:

```
   (x_k, y_k,q_k) = ( (x-k*y), y, (q+k));
```

Also both of the following conditions must be true:

```
 ge(x_(k-1), y_(k-1)) = TRUE;
and   ge( x_k,y_k) = FALSE;
```

```
 Hence the following must be true:
condition 1:    (x -(k-1)*y) >= y;
condition 2:    (x-k*y) < y;
```

The effect of executing the operator o.quotient is thus given by:

```
  quotient (x,y) = ( k, (x-k*y))
   where: k is defined by the two equations above.
```

The operator o.quotient accepts two integers (x y) and returns the quotient k and the remainder (x - k*y); i.e. x = r + k*y.

Since our remainder 'r' is given by (x - k*y), condition 2 above is a straightforward expression of the second assertion (r < y). The first assertion (x= r+ k*y) is trivially seen by substituting (x-k*y) for r. Finally, the third assertion, i.e. (r > 0) is seen from condition 1 which states that: x - (k-1)*y >= y. From this it follows that:

```
 x - k*y + y >= y;
i.e. r + y >= y;
i.e. r >= 0; since we assumed initially that y > 0.
```

Thus all the three assertions are proven to hold true. Hence the operator o.quotient is said to be verified.

Hoare (Hoare 1969) has discussed the proof procedure of an essentially identical program within the context of imperative languages. What is interesting is the fact that in constructing the proof of the program, Hoare assumed absence of side effects in evaluation of assignment statements and condition expressions. Farther it was assumed that the programs do not employ jumps, pointers and name parameters. Notice that these constraints are unlikely to be obeyed by most programs in the imperative languages. Thus it appears that it will be difficult to verify many, if not most, programs employing Hoare's methodology. In short, Hoare's methodology will not be applicable to verify programs which employ side effects. Most programs in imperative languages however do employ side effects. On the contrary, the EML language allows locally controlled side effects; also it provides a verification procedure for a syntactically valid programs in the language in a very straight-forward manner.

Note that, while the EML language guarantees a simple way to express the effect of loop computation in the form of an expression involving the meta function <u>loop update</u>, it by no means assures that every syntactically valid loop statement will converge to some specific termination during actual execution. Termination conditions should be the property of each individual loop rather than the property of a programming language. Thus, in EML as well as in other programming languages, termination properties of a loop have to be decided by independent means. Of course in the example of o.quotient, it is almost trivial to prove that the loop will terminate if both x and y are positive integers. With successive iterations the value of k monotonically increases and value of (x - k*y) monotonically decreases. Thus after a finite number of iterations the condition ' (x -k*y) > y' will be false and the loop will terminate. This part of the verification process is essentially equivalent to the verification process of other languages, except that monotonicity is usually easier to guarantee due to the controlled nature of side effects.

5.2.2 Example 2:

We will now discuss the verification of a program intended to compute the greatest common divisor of two positive integers using the Eucleadian algorithm (Djikstra '76). Djikstra has discussed extensively the difficulty of finding an appropriate loop_invariant for the loop in the Eucleadian algorithm. The program written in 'C' is in figure 5.3.

Figure 5.3
A Program For Determining The Greatest Common Divisor

```
 int  p;
 main ()
     { int X,Y, Z;

       scanf ( "%d %d", &X, &Y);
       Z = gcd ( X, Y);
       printf ( "%d", Z);
     }

gcd ( X,Y)
int   X,Y;
    {  while ( ( X != Y) && ( X != 0)
                    && ( Y != 0) )
              { if ( X > Y) then X = X - Y;
                 else Y = Y -X;
              }
        p = X*X; /* side effect */
        return ( X);
    }
```

The same algorithm is implemented in the EML operator o.gcd in figure 5.4. First we discuss the verification of the program in 'C' to contrast it with the ease of verification of the corresponding EML program. The program in 'C' uses the procedure "gcd' to compute the greatest common divisor. The program is implemented using a procedure to illustrate the difficulty of verifying a program using a procedure with side-effects. The statement "P = X*X;" in the procedure gcd causes the side-effect, since it alters the value of the nonlocal variable 'P'. Admittedly, this is a mindless addition to the procedure and does in no way contribute to the derivation of the greatest common divisor. Neither is this addition a part of the Eucleadian algorithm par se. Nonetheless, the procedure gcd is syntactically valid and the example will serve to illustrate the difficulty one faces in transforming a procedure of an imperative language to an expression capturing the effect of its execution.

Figure 5.4

```
Operator o.gcd

o.gcd
{ vars: v.int1, v.int2;

  receive () =   v.int1, v.int2;
  loop
  [ if (and(ne(v.int1,v.int2) ,
        and(ne(v.int1, 0),
        ne(v.int2, 0))))
        o.lgcm( v.int1, v.int2)
          =     (v.int1, v.int2);];
  send ( v.int1);
}
```

Table 5.3

Environments of operator o.gcd(x,y)

objects	Environments		
	Env(0)	Env(1)	Env(2)
v.int1	E	x	x_k
v.int2	E	y	y_k
c.zero	0	0	0

```
   (x_k, y_k) = loop_eval( ENV(1),
                  (and(ne(v.int1,v.int2),
                    and(ne(v.int1,0), ne(v.int2,0))),
                      (v.int1,v.int2), lgcm);
```

In transforming the procedure "gcd" to an expression we ought to be able to transform the effect of its "while" statement to an expression so that we could relate the values X, Y, after the loop termination to their values before the initiation of the loop. It seems there is no simple way to intuit such an expression. In fact, we have, so far failed to derive such an expression within the context of semantics of the imperative language. For the purposes of this discussion however, let us assume that such an expression exists; also assume that values of X,Y before and after the termination of the loop are (x y) and (x_k y_k) respectively. Thus we could reason that the procedure gcd receives the values x,y and would return the value x_k. Therefore the effect of executing "gcd" could be expressed by the relation: x_k = gcd(x,y). However, this would still be wrong since this expression would not reveal the fact that the value of the variable p has been modified to (x_k*x_k). In essence, it seems to be difficult, if not impossible to state the effect of executing the procedure gcd in the form of an expression. In contrast, we show below that the effect of executing the EML operator o.gcd can be derived in an algorithmic manner.

An EML operator computing the greatest common divisor of two integers (x y) such that (x > 0) and (y > 0), will be considered verified if the operator returns a value y_k such that the following relations hold true:

```
x = m* x_k + y_k;
y = n* x_k + y_k;
x_k = p* y_k;
 where  x_k, y_k, m, n, p are integers such that:
        (a)  x_K > 0; y_k >0;
        (b) m >=0; n >= 0;  p > 0;
```

In our implementation the operator o.gcd invokes the operator o.lgcm. The operators o.gcd and o.lgcm are described in the figures 5.4 and 5.5. The environments for these operators are in tables 5.3 and 5.4 respectively.

Figure 5.5

```
Operator o.lgcm

o.lgcm
{ vars : v.int1, v.int2,
                v.next1, v.next2;

  receive() =   v.int1, v.int2;
  copy (v.int1,2) =   v.int1, v.next1;
  copy (v.int2,2) =   v.int2, v.next2;

  if ( gt( v.int1,v.int2))
       sub(v.int1, v.int2) =  v.next1;
  sub ( v.int2, v.int1)   =  v.next2;

  send ( v.next1, v.next2).
}
```

Table 5.4

Environments of o.lgcm (x,y)

objects	Environments					
	ENV (0)	ENV (1)	ENV (2)	ENV (3)	ENV (4)	ENV (5)
v.int1	E	x	x	x	x4	x5
v.int2	E	y	y	y	y4	y5
v.next1	E	E	x	x	p4	p5
v.next2	E	E	E	y	q4	q5

```
(x4 y4 p4 q4) = map_value ( ENV(4),
                    (v.int1, v.int2, v.next1, v.next2)
        where:
        { ENV(4) = conditional_update( ENV(3),
                (gt(v.int1,v.int2)), (v.int1,v.int2),
                   (v.int1,v.int2), (v.int1,v.int2),
                     sub, (v.next1));
        };

(x5 y5 p5 q5) = map_value( ENV(5),
                    (v.int1, v.int2, v.next1, v.next2)
        where:
        { ENV(5) = simple_update ( ENV(4),
                   (v.int2,v.int1), sub, (v.next2));
        }
```

The effect of executing o.lgcm(x,y) is given by:

```
  lgcm(x,y) = send_update( ENV(5), (v.next1, v.next2));
```

For inputs such that (x>y) or (x <= y) the environment table 5.4 may be easily reworked to produce the following:

```
  lgcm(x y) = if ( gt(x y)) then ( (x-y), y);
              if(not(gt(x y)) then ( x, (y-x));
```

Now we can transform the operator to o.gcd to its equivalent function and analyze its the properties. From figure 5.4 and table 5.3 equivalent function of o.gcd is defined to be:

```
 gcd(x y) = send_update( ENV(2), (v.intl))
            where ENV(2) is obtained from table 5.3.
```

The value_list (x_k y_k) in table 5.3 is obtained by evaluating the expression :

```
  loop_eval( ENV(2),
            (and(ne(v.intl,v.int2),
              and(ne(v.intl,0), ne(v.int2,0))),
              (v.intl,v.int2), lgcm);
```

This expression is evaluated by applying induction (see chapter 4) in the following manner. Assume that variables v.intl,v.int2 have values x_i,y_i during the i_th iteration of the loop_eval function. Then we find the following:

```
  (x0 y0) = (x y);
  (xl yl) = lgcm(x0  y0);
     ...
  (xk yk) = lgcm(x_k-1  y_k-1);
```

Some properties of the equivalent function gcd can be gleaned in the following manner. Assume that initially: (x > y) and that after k iterations through the loop x_k <= y_k. With this restriction on the input we get the following:

```
  (x0  y0) = (x y);
  (xl yl) = lgcm(x y)
          = ( (x-y), y); since we have assumed x > y;
  (x_k-l y_k-l) = ((x -(k-l)*y) y);
  (x_k y_k) =  ( (x- k*y) y);
```

If the loop terminates after the k_th iteration then either or both of the following relations must be true.

```
  and(ne(v.intl,v.int2),
   and(ne(v.intl,0), ne(v.int2,0))) = FALSE;
 i.e either    x_k = y_k;
        or     x_k = 0;

  If x_k = y_k,
   then    x - k*y = y;
   i.e. x = (k+1)* y;    ....           (1)

  If x_k = 0
   then  x- k*y = 0;
    i.e  x = k*y;        .....          (2)
```

In other words x is an integral multiple of y; i.e. y is the greatest common divisor of x and y. Since x_k = y_k = y, the value x_k returned by the operator gcd would represent the greatest common divisor of x and y. Note that if in the input the relationship of x and y were reversed, i.e. if in the input (x <= y) then also the same reasoning would hold.

Both of the relations (1) or (2) above also satisfy the three assertions described in the beginning of the section. This is trivially seen by equating y_k to y and adjusting the constants m,n,p as follows:

```
   x = k* x_k + y_k;
   y = 0* x_k + y_k;
   x_k = 1* y_k;
```

It is possible that after the k_th iteration the loop does not terminate but x_k becomes less than y_k i.e. x_k < y becomes true. There are two possible cases:

```
  x_k = 1;
  or 1 < x_k < y;
```

In each case looping would continue and the effect could be described by the recurrence equations below.

```
 (x0  y0) = (x_k y);
 (x_k' y_k') = ( x_k  (y - k'*(x_k)); since x_k < y;
```

If x_k = 1 then it is obvious the loop would terminate after another k' iterations. The termination condition (y - k'*1) = 1 will be satisfied no matter what the value of y is since k' is monotonically increasing with every iteration. In this case the value returned by the operator gcd is 1 which is a divisor for any two integers. It is obvious that if the operator returns 1, then the assertions are satisfied as well.

If 1< x_k < y, then looping would continue with the property that at each iteration one of the variables decrease while other does not. This goes on till the decreasing variable becomes larger than the non_decreasing one. At this time the variables switch role; i.e. the variable which was decreasing before becomes non_decreasing and the variable which was non_decreasing starts decreasing. None of the variable however can decrease below 1 when the loop is guaranteed to terminate.

The more general case i.e. the case in which the variables switch role can be analyzed in the following manner. Let the initial values of v.int1 and v.int2 be x(0) and y(0) respectively and let (x(0) > y(0)) be true. Let after p0 iterations v.int1 have the value x(1) such that (x1 < y0) be true. Now the variables switch role; i.e now onwards value of v.int1 remain constant and value of v.int2 continue to decrease till another switch occurs. Let after q0 iterations v.int2 have the value y(1) such that (y(1) < x(1)) be true. The switchings would go on till at the beginning of a switch the values of v.int1 and v.int2 be x(k) and y(k) such that either (a) x(k) is an integral multiple of y(k); or (b) x(k) = 1; or (c) y(k) = 1. We have argued that under these conditions the loop will terminate without any farther switching of role. If x(k) is an integral multiple of y(k) then y(k) is the greatest common divisor of these two integers. We will now use inductive reasoning to show that y(k) would also be the greatest common divisor of x(0) and y(0). The values of x(i), y(j) and the relationship among them at the beginning of every switch are described below:

```
( x(0)   y(0) );  (y(0) < x(0))  /*initial condition*/
( x(1)   y(0) );  (x(1) < y(0)); (x(0)=p0*y(0) +x(1));
( x(1)   y(1) );  (y(1) < x(1)); (y(0)=q0*x(1) +y(1));
( x(2)   y(1) );  (x(2) < y(1)); (x(1)=p1*y(1) +x(2));
( x(2)   y(2) );  (y(2) < x(2)); (y(1)=q1*x(2) + y(2));
         ......
         ......
(x(k-1) y(k-1)); (y(k-1) < x(k-1));
                      (y(k-2)=q_(k-2)*x_(k-1) + y(k-1));
(x(k)  y(k-1));  (x(k) < y(k-1));
                      (x(k-1)=p_(k-1)*y(k-1) +x(k));
(x(k)  y(k));    (y(k) < x(k));
                      (y(k-1)=q_(k-1)*x(k) +y(k));
(x(k+1) y(k));  (x(k+1) = y(k));
                      (x(k) =p_(k+1)*y(k) + 0 );
```

The above relationships show that y(k) is the greatest common divisor of (x(k) y(k)). Now we shall apply induction to prove that y(k) is the greatest common divisor of each preceding pair; i.e. y(k) is the greatest common divisor of the pairs (x(k), y(k-1)), (x(k-1), y(k-1)) ... (x(2) y(1)), (x(1) y(1)), (x(1) y(0)), (x(0) y(0)). We know from the above:

```
  x(k) = p_(k+1)*y(k);
  We know from above :
  y(k-1) = q_(k-1)*x(k) + y(k);
 i.e. y(k-1) = m*x(k-1) + y(k)
        where the integer m = q_(k-1).
```

Since y(k) is the greatest common divisor of (x(k) y(k)), y(k) must also be the greatest common divisor of (x(k) (m*x(k) +y(k))); i.e. y(k) must also be the greatest common divisor of (x(k) y(k-1)). Similarly by employing simple algebra we can reduce x(k-1) and y(k-1) to expressions of the following form:

```
  x(k-1) = n1*x(k) + y(k);
  y(k-1) = m*x(k) +y(k);
```

Since y(k) is the greatest common divisor of (x(k) y(k)), y(k) must also be the greatest common divisor of ((n1*x(k) +y(k)), (m*x(k) + y(k)) i.e. y(k) must be the greatest common divisor of the pair (x(k-1) y(k-1)). Similarly we can show y(k) is the greatest common divisor of the pairs (x(k-2) y(k-1)), (x(k-2) y(k-2) .. (x(1) y(1)), (x(1) y(0)), (x(0) y(0)). In general the proof depends on our ability to express every (x(i) y(j)) pair in the form ((n_i* x(k) + y(k)) , (m_j*x(k) + y(k))). Thus y(k) is the greatest common divisor of each such pair as long as y(k) is the greatest common divisor of (x(k) y(k)). Hence y(k) is the greatest common divisor of (x(0) y(0)) i.e. of the initial values of v.int1 and v.int2. Note that this analysis shows that the values of x(0),y(0), x(k), y(k) are related by expressions of the following form:

```
  x(0) = m*x(k) + y(k);
  y(0) = n*x(k) + y(k)
  x(k) = p* y(k);
```

Thus all the assertions specified for the operator o.gcd are shown to be true. Hence the operator o.gcd is said to be verified.

We could mention here again that the application of inductive reasoning to show that the loop terminates and that it terminates to provide the greatest common divisor has nothing to do with the EML language as such. If we could generate the function for the loop statement by any other means, the application of induction would be as easy or as hard, depending on the that an expression for the function computed by the loop. The beauty of EML lies in the fact, that the function computed by the loop can be obtained by simple application of the semantic specification on the source code.

The main point of this discussion is this: Given an EML operator we can transform it to an expression by following the semantic specification of the EML language. The expression may be complex reflecting the complexity of the algorithm implemented by the operator. However, by assuming suitable constraints on the input set, many useful properties may be deduced from the expression. We have shown above that such may not necessarily be the case for programs implemented in von Neumann languages, because of the presence of uncontrolled side effects. Note, for instance that if o.gcd contained a statement " sqr(v.intl) = v.P", our environment table would have appeared wider but our analysis would have remained unchanged.

5.2.3 Example 3:

In this example, we approach the verification of the operator o.factorial which employs recursion. This operator is designed to accept a positive integer as its input and return the factorial of the integer as its output. The operator o.factorial is shown in figure 5.6. Its environments are shown in table 5.5. The expressions relating the values in some of the environments to the values in the preceding environment are shown in table 5.6. From table 5.5, the effect of executing o.factorial(n) is given by:

```
  factorial( n) = send_update ( ENV(6), (v.fact));
```

We can derive 3 specific solutions for factorial for cases: n > 0, n=0, n <0. For these inputs, the environments in table 5.5 can be reworked to produce the following specific solution for the equivalent function factorial:

```
  factorial(x) = if (x > 0) then x*factorial(x-1);
               = if (x = 0) then 1;
               = if (x < 0) then error;
```

Like the loop statements discussed above, we should note that in general there is no guarantee that a recursive function on execution will terminate. The recursion termination, like loop termination, must be proved, if necessary, by independent inductive reasoning. If the domain of x is restricted to positive integers, then it is easy to prove that the above recursive function will finally terminate.

Figure 5.6

Operator o.factorial

```
o.factorial
{  vars :v.N, v.M, v.fact, v.next_fact,
         v.next_num;

   receive () = (v.N);
   copy (v.N, 2) = (v.N, v.M);
   if (eq(v.N,0)
      copy(1,1)  = v.fact;
   if (gt( v.N, 0)
      sub( v.M, 1) =v.next_num;
   o.factorial(v.next_num)
                    = v.next_fact;
   mult(v.N, v.next_fact)
                    = v.fact;
   send (v.fact).
 }
```

Table 5.5

Environments of o.factorial (n)

object names	Environments						
	ENV (0)	ENV (1)	ENV (2)	ENV (3)	ENV (4)	ENV (5)	ENV (6)
v.N	E	n	n	n	n	n	n6
v.M	E	E	n	n	m4	m4	m4
v.next_ num	E	E	E	E	n_n4	n_n4	n_n5
v.next_ fact	E	E	E	E	E	nf5	nf6
v.fact	E	E	E	f2	f2	f2	f6
c.zero	0	0	0	0	0	0	0
c.one	1	1	1	1	1	1	1
C.two	2	2	2	2	2	2	2

```
n6 = if (none_empty(n,nf5)) then (E) else ( n);
f2 = if (eq(n,0)) then (1) else (E);
m4 = if ( gt (n,0)) then ( E) else (n);
n_n4 = if (gt (n,0)) then (n-1) else (E);
nf5  = if (none_empty( n_n4)) then ( factorial(n_n4))
                             else ( E);
n_n5 = if (none_empty (n_n4)) then (E) else (n_n4);
nf6  = if (none_empty (n, nf5)) then (E) else (nf5);
f6   = if (none_empty (n, nf5)) then ( n* nf5)
                             else (f2);
```

5.2.4 Example 4:

The examples in this section relate to non_terminating loops. The operators o.non_terminating_1, o.non_terminating_2 and o.may_not_terminate are described in figure 5.7.

Figure 5.7
Description of operators containing
non_terminating loops

```
o.non_terminating_1
{  vars   v.x;
   receive () = (v.x);
   loop
   [ if (true) o.add_one(v.x) = (v.x);]
   send (v.x).
}

o.non_terminating_2
{  vars v.x, v.y, v.z;
   receive () = (v.x, v.y, v.z);
   loop
   [ if (gt(v.y, v.z))
        o.add_one(v.x) = (v.x);]
   send(v.x).
}

o.may_not_terminate
{ vars  v.x;
  receive () = (v.x);
  loop
  [ if (gt(v.x,5))
       o.add_one(v.x) = (v.x);
  send (v.x).
}

o.add_one
{  vars  v.x, v.x1;
   receive () = (v.x);
   add (v.x, 1) = v.x1;
   send(v.x1).
}
```

All these there operators invoke the procedure o.add_one in their loop statement. The three operators employ different condition expressions in their loop statement. The effect of executing o.add_one is easily derived to be:

```
add_one(x) = (x+1);
```

From the definition of the operators we can easily intuit the following: the operator o.non_terminating_1, if invoked will run for ever since its the condition expression of its loop statement always returns true. Note the condition expression in a loop statement actually represents condition for loop iteration. Similarly, in the operator o.non_terminating_2 there is no overlap of variables between condition expression and procedure expression. Hence in this case, successive iteration of the loop will not influence the result of evaluating the condition expression. Thus in this operator if the loop statement executes once it will execute for ever, or it will not execute at all. In the third example the continuation of the loop is dependent on the initial value of input variable. If the condition 'gt(v.intl,5)' is initially true then it will continue forever since successive iteration of o.add_one monotonically increases the value associated with v.intl. Hence the condition 'gt(v.intl ,5)' will be true forever and the loop will never terminate. If on the other hand, the condition is initially is false then the loop statement will not execute at all. It should be noted that the possibility of infinite loops in the first two operators can be inferred by simple syntactic examination of the loop statements in these operators. Thus it is possible that a suitable compiler for the EML language could detect these two types of nonterminating loops at compile time.

CHAPTER 6

DISCUSSION

6.1 Program Verification In General: EML vs. Imperative Languages:

We will now compare some features of EML with those of imperative languages to illustrate why it may be difficult to transform programs of the latter to expressions. In transforming the operators in the EML languages we have essentially used the technique of forward substitution, i.e. the values resident in some state are used to generate values of variables in the next state. This is possible because of the following features of the EML language:

(1) The EML language has a very simple control flow. The syntax of the EML language does not allow any jump or backward branching in the control flow during execution.

(2) The _loop_ statement in EML is an atomic operation the meaning of which is well defined by the function _loop_-_update_.

(3) A value is never bound to more than one name at any time; also the procedures never share either data_space or name_space with other procedures. As a result, the effect of executing a statement on the environment containing the statement is very easily deduced.

This method is not quite suitable for all syntactically valid programs in imperative languages because:

(a) These languages often allow complex control flow including backward branching. It is difficult to apply forward substitution in these cases.

(b) Often in these languages it may be difficult to transform loops into expressions.

(c) These languages allow a value to be associated with more than one name; also the procedures often share both name_space and data_space with other procedures. Hence if the value associated with one variable name is changed, then by side effect, states of other variables in the same or different contexts may get changed. Hence the effect of executing a statement on the environment containing the statement is not easily deduced.

We shall first discuss the difficulty of transforming the loop statements in the imperative languages. Let us consider the program fragment shown below:

```
while ( x > k)
{  x = x -1;
   y = y + 1;
}
```

In order to transform the while loop we need to relate the values of X and Y after the execution of the loop to values of the same variables before the execution of the loop. Assume the values of X to be x and x_k before and after the execution of the loop; similarly let values of Y be y and y_k. We are to capture the effect of executing the effect of the 'while' statement on the environment containing the statement. If we could be sure that the execution of this statement does not affect the state of any other variable in any other context, we could express the effect of executing the statement by relating x_k and y_k to x and y as follows:

```
x_k = if ( x > k) then k else x;
y_k = if ( x > k) then ( y + ( x - k)) else y;
```

Notice that as long as the threat of side effects exist, there appears to be no algorithmic way to capture the effect. We have to apply our intuition to achieve it. There is no guarantee that such expressions will be intuitively obtained for all syntactically valid loop statements. The loop-statement in the gcd procedure discussed above (figure 5.4) is a case in point. We suspect that whenever there is a branch within a loop statement, it will be difficult to represent the effect of loop computation as an expression.

In the traditional approach loop computation is verified by proving the existence of a loop-invariant for each loop (Manna '77, Misra '78). The problem of finding a loop invariant is theoretically unsolvable (Wegbreit '74). Misra (Misra '78) indicates that if a loop computes a function then it must have a loop-invariant; however there is no certainity that every syntactically correct loop statement will compute a function. Dunlop and Basili (Dunlop and Basili '82) have discussed an approach of loop verification based on the functional correctness of a loop. In this method, one would determine the function computed by a loop and then attempt to verify its properties. They (Dunlop and Basili) have shown that the problem of finding a loop invariant and the problem of finding the function computed by a loop are theoretically equivalent. Hence the method of using loop-invariant for loop verification is no better than the method of transforming a loop to a function and then determining its functional correctness. The difficulty is that in the imperative languages there is no algorithmic way to find the function computed by syntactically correct loop statement. It is here that EML provides an advantage. The semantics of EML specifies the function computed by syntactically correct loop statements using the meta_functions like loop update and others described in chapter 4. Thus transformation of loop statements into the functions they compute is algorithmic. Once the function is obtained, inductive reasoning may be applied to determine the properties of the loop. In order to avoid the difficulty of loop verification Djikstra suggests that the loop invariant be determined before the loop statement is designed and that the loop statement be composed to reflect the loop-invariant. This of course becomes a matter of style. It will not be quite trivial for a compiler to determine if a loop statement has been composed to satisfy a loop-invariant. On the other hand, the syntax and semantics of EML assures that every syntactically correct loop statement can be replaced by a function expression, though this expression may not in fact define a function. It is not left to be a matter of good programming practice. It ought to be noted that in EML the loop_statement is an atomic statement. The operator in the procedure expression of the loop statement, is like any other user defined operator. We have already shown that a syntactically correct EML operator may be easily transformed to its equivalent function. Thus the programming process in EML requires the programmer to conceptualize loops so that they compute a function. Misra (Misra '78) has indicated certain conditions which a programming language should fulfill so that a loop statement in the language would compute a function. These include the following: initially the local variables in the loop statement (procedure) should be undefined; a local variable may not be accessed before it is assigned a value. It is interesting to note that EML language enforces both of these constraints.

We now return to the issue of side effects during the execution of a procedure. It has been stated that if during execution, a procedure modifies the value(s) of one or more nonlocal variables, then a side effect is said to occur. It is well known that if a procedure alters the state of memory beyond its own context, then the procedure may not be treated as a function. Hence the effect of executing a procedure may not be represented by an expression. In EML, a procedure is never allowed to access nonlocal memory. A procedure Pi invoking another procedure Pj, merely view the procedure Pj as a mapper of a value_list to another value_list. The procedure Pi, manages its own environment by the rules of consumption and production described in this paper. The rules for managing the environment are independent of the procedure being invoked. Since an EML procedure never alters the state of nonlocal memory, the effect of executing an EML procedure may be equated to a function. This allows the development of dual view for EML procedures. Internal to each procedure there is the concept of environment and state; external to the procedure it may be viewed as a function.

These facts viz. the ability to view EML operators as functions, the ability to express a loop computation as an expression, and the ability to apply forward substitution make it possible to describe the meaning of executing an EML operator in the form of an expression. This expression embodies all properties of the procedure. We may have a good chance to predict the properties of an operator by analyzing its environments and the expression. In general this is not possible for a syntactically valid program in the imperative languages. We hope, therefore, that the verification of programs ought to be easier if the programs were coded in the EML language.

Why did we choose to transform programs to expressions as the method for verification of EML programs? Currently there are two models for verifying programs in imperative languages (Basili 1980, Dunlop and Basili 1982). These are the axiomatic model proposed by Hoare (Hoare 1969) and Floyd (Floyd 1967) and the functional model proposed by Mills (Mills 1975). The goal of functional model is to establish the functional correctness of a program. This method tries to describe the effect of executing the program on the data as an expression or function relating the input values to output values. The functional model of program verification is largely defeated in the context of von Neumann languages because of side effects. Side effects often make it impossible to describe the effect of executing a procedure in terms of an expression involving its input values. In the EML language side effects are strictly controlled. The semantics of the EML language allows an EML procedure to be transformed to a function expression using a simple algorithm. Once described as a function expression, the properties of the program can be analyzed and predicted with mathematical rigor. The necessity for being clever or inventive largely disappears. Therefore it is simple to verify EML programs following the functional model.

The axiomatic model of program verification requires an inductive assertion on each loop as well as input-output specifications for the program (Floyd 1967). An assertion itself is a relationship between states of variables involved. Using the methods of predicate calculus, each assertion indicates what should be true about the state of variables at the particular point in a program where the assertion appears. Note that the method of verification utilizing invariant assertions merely seek to verify some specified relationship among some chosen variables. EML on the other hand specifies the value of each variable (as an expression) at each stage of execution. Creating or inventing appropriate inductive assertion, loop invariant assertions included, is a very difficult job. There is no known algorithm for constructing the suitable assertions for all programs or loops. Inventing the appropriate assertions often require intuition and outright cleverness. Compared to these, verification of EML programs can be done in a much more routine manner following the functional model.

In our view the difficulty of verifying a program in von Neumann language is in effect a result of complex semantics employed in these languages. Even symbolic execution of programs in these languages are complicated by their complex scope rules and side effects (Howden 1977). Compared to these efforts, the transformation of EML operators are simple and straightforward. The simplicity of the transformation of EML is a reflection of the simple semantics employed by the language.

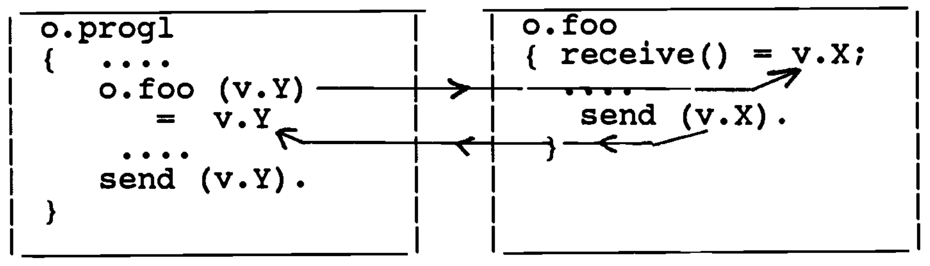
6.2 Dual view of operators:

The procedures in von Neumann languages present a state_based view of the execution of a program. The procedures have variables, i.e. addresses in memory where values are stored. They allow the variables to assume different values during execution corresponding to change of state. We have argued in the beginning of this paper that because of the facilities which allow procedures to share data space and name space with other procedures (including the main procedure), it is not possible to develop a functional view of von Neumann procedures. On the other hand functional languages in their pure form portray a state_less machine. The "read-eval-print" model of pure Lisp (McCarthy 1960) is an example of this model. The functional machine evaluates an expression and delivers a value. It does not recognize the notion of memory to store a value. Thus it cannot represent changes of states as computation proceeds. The operators in EML on the other hand provide a dual view. Internally the operators maintain the notion of variable names and binding between values and names. The same name can be bound to different values at different stages of computation reflecting changes of state as execution proceeds. This aspect has been adequately demonstrated in the environment table of each operator. Also the definition of operators in EML are such that, an operator may not share data space or name space with any other procedure. This clear separation of environments of an operator from all other operators allow us view an operator as a function from environments external to the operator. THE NET EFFECT IS THAT OPERATORS IN EML CAN MODEL BOTH THE STATE BASED VIEW AND THE FUNCTIONAL VIEW DEPENDING ON WHETHER ONE IS VIEWING THE INTERIOR OR THE EXTERIOR OF THE OPERATOR. The two views are clearly separated and yet they are not mutually exclusive. In fact the clear separation of the two views of EML operators have allowed us to develop the simple formalism for verifying EML program.

6.3 Efficiency of data transfer between contexts:

We have mentioned that in EML an operator shares neither name-space nor data space with other operators; yet data values are transferred between contexts. In functional environments as well as in data flow languages the only method of data transfer is to copy the data from one context to another context. There appears to be no exception. The method can become quite inefficient when complex data structures like arrays, graphs or data bases are to be transferred to a different context. In our opinion the EML provides a quite inexpensive solution for data transfer between contexts. In EML, when a variable is consumed, the association between its name and its value is broken. The same value is then associated with the target name in the destination operator. In the source operator the name of the consumed variable is associated with the empty value. Thus there is no need to physically copy data from one environment to another. This concept should improve performance by reducing the need to copy and thereby allowing data spaces to be used more efficiently. The figure 6.1 illustrates this concept.

Figure 6.1
Data Transfer between operators



```
o.progl                      o.foo
{  ....                      { receive() = v.X;
   o.foo (v.Y)                  ....
     =  v.Y                     send (v.X).
   ....                      }
   send (v.Y).
}
```

Assume that the operator o.progl invokes the operator o.foo with the value of a data base v.Y. The EML machine breaks the association between the name v.Y and the value of the data base and associates the same value to the name v.X in the operator o.foo. After the operator has processed the value, it is reassociated with the name v.Y in o.progl. During the time o.foo was active, the name v.Y in o.progl was bound to the value E̲. Thus no matter how large the data is, the cost of transferring it from environment to environment is quite negligible both in terms of space and time. In the environment of the functional and of data flow languages the same operation would require the data to be copied twice. Note that the EML machine does not copy a value unless it is specifically asked to by invoking the 'copy' operator. In other words the EML restricts access to a value by only one operator at time. Two operators may not access the same value at the same time. This is possible because, we assume that in EML, a name is bound to a value and we let the EML machine take care of the physical location of the value. In von Neumann languages however, the concept is somewhat different. A variable name, in these languages is bound to an address in the memory and there is no control over the number of names which may be bound to an address. Therefore in the von Neumann Languages, the only mechanism to transfer data from one context to another is to copy the data to the destination environment.

6.4 Position of EML in the family of programming languages:

The programming languages may be broadly divided in two groups: the applicative languages and the imperative languages. We will discuss the relationship of EML to these classes of languages, in general.

The applicative languages are built around the notion of 'values' and mapping of a set of values to another set of values via the application of functions. These languages have no notion of variables as containers of values; nor have they any notion of memory or state. The effect of executing a function in the applicative languages is to reach new values from input values; the output and input values are related via an equation involving the function. The effect of executing an EML operator may be viewed at two levels: the effect on values and the effect on variables (or memory). Viewed at the level of values only, an EML operator appears more like a function in the applicative languages since the output values generated by executing an EML operator are related to its symbolic input values by means of an equation involving the equivalent function of the operator. Aside from generating new values from existing values, the execution of an EML operator changes the state of memory as well. The execution of an EML operator will alter existing values in all variables in the procedure expression invoking the operator. This is in contrast to the notion of functions since the application of a function does not alter any member in its domain or range. Thus the EML may be viewed as an applicative language to which the notion of environment has been added. The notion of environment is captured and localized in the syntactic construct procedure expression.

The imperative languages, whether they are D_structured, L_structured or block-structured (Kosharaju '74) are all built around the notion of variables, memory and assignment operator. Although the EML language recognizes the first two, it has no notion of an assignment operator. On the other hand, the imperative languages have no notion of consumption and production of values, the notions which are central to understanding the EML language. The procedure expression in the EML is the closest construct to the assignment operators in imperative languages. In the imperative languages, the execution of an assignment statement will probably , but not necessarily, change the value(s) of one (or more) variables in the memory. In EML the execution of a procedure expression will surely change the values of all variables referred to in the procedure expression. There is another significant difference between the assignment statements and the procedure expression. At run time, if an assignment statement is accessed then the statement will definitely be executed no matter what the values of the variables are. The executability of an assignment statement is never sensitive to values of variables occurring in the assignment statement. In contrast, in EML, the executability of a procedure expression is sensitive to the run time values of its consume and produce variables. Even if the statement containing the procedure expression is accessed, the procedure expression may not be executed if the readiness condition is not satisfied.

Whether or not the EML language will turn out to be a useful tool for production and maintenance of software remains a conjecture at this time. These issues can be decided only after considerable experience has been gathered in the use of the language. Certain features of the language however deserve comment at this time. The notions of consumption and production of variables allows the variables of EML to be viewed as objects in the physical world. In the physical world, an object participating in the production of other objects is itself consumed in the process. We think that the view of variables implemented in EML is more natural than the view of variables in the imperative languages. Hopefully the more natural view of variables will help reduce the semantic gap between the application domain and programming domain (Bergland '81) and thus aid the design of software. The meaning of executing an EML operator is not influenced by the calling environment. This property should be useful in developing an incremental programming environment using EML. Because of the simpler semantics, programs in EML ought to be easier to understand, maintain and modify compared to programs in the imperative languages. A number of studies have indicated that higher the number of branches and knots in the control flow of the program, more difficult is it to understand, maintain and modify programs (McCabe '76, Chen '78). The syntax of the EML language does not allow for any branch or jump to any statement other than the immediate next statement. Each statement bounded by the receive and send statements must be accessed in a linear sequence. The control flow of an EML program thus cannot have any branch or knot. This would suggest that understanding and modifying EML programs should be easier. Since the maintenance cost of a software is approximately 50% of its total cost, this feature of EML is quite attractive and deserves closer scrutiny.

BIBLIOGRAPHY

[1] W.B.Ackerman, "Data Flow Languages", Computer vol 15(2), February 1982.

[2] T. Agarwala and Arvind, "Data Flow Systems", Computer vol 15(2), February 1982.

[3] Arvind and Kim P. Gostelow, " The U-interpreter", Computer vol 15(2), 1982.

[4] J. Backus, "Can Programming Be Liberated From The von Neumann Style? A Functional Style And Its Algebra Of programs", Commun. Ass. Comput. Mach, vol 21, pp 613-644, Aug 1978.

[5] V. R. Basili and R.E. Noonan, "A Comparison Of Axiomatic And Functional Models Of Structured Programming", IEEE Trans. Software Eng. vol SE-6 (6), Sept 1980.

[6] G. D. Bergland, "A Guided Tour Of Program Design Methodologies", Computer vol 14, pp 13-37, 1981.

[7] E. T. Chen ,"Program Complexity and Programmer Productivity", IEEE Trans. Software Eng. vol SE-4, pp 189-194, May 1978.

[8] N. Dershowitz and Z. Manna, " The Evolution Of Programs: Automatic Program Modification", IEEE Trans. Software Eng. vol SE-3 (6), Nov 1977.

[9] E. W. Djikstra, "A Discipline Of Programming", Englewood Cliffs, N.J, Prentice-Hall, 1976.

[10] D. D. Dunlop and V. R. Basili, "A Comparative Analysis Of Functional Correctness", ACM Computing Surveys. vol 14(2),pp 229-245, 1982.

[11] R. W. Floyd, "Assigning Meaning To Programs", Proc. Symp. Applied Math. vol 19, J.T Schwartz, Ed. Amer. Math. Society, pp 19-32, 1967.

[12] J. W. Goodwin, "Why Programming Environments Need Dynamic Data Types", IEEE Trans. Software Eng. vol SE-7 (5), Sept 1981.

[13] P. Henderson ,"Functional Programming Application and Implementation", Prentice Hall, 1980.

[14] C. A. R. Hoare, "An Axiomatic Basis For Computer

Programming", Commun. Ass. Comput. Mach, vol 12 (10), October, 1969.

[15] W. E. Howden,"Symbolic Testing And The Dissect Symbolic Evaluation System", IEEE Trans. Software ENg., vol SE-3, 1977.

[16] R. Kosharaju, "Analysis Of Structured Programs", J. Comput. And System Sciences, vol 9(3), 1974.

[17] T. J. McCabe, "A Complexity Measure",IEEE Trans. Software Eng., vol SE-2, December 1976.

[18] J. McCarthy, "Recursive Functions Of Symbolic Expressions and Their Computation By Machines", Comm. ACM, 3(4), 184-195.

[19] H. D. Mills, "The New Math Of Computer Programming", Commun. Ass. Comput. Mach. vol 8, January 1975.

[20] J. Misra, "Some Aspects Of The Verifications Of Loop Computations", IEEE Trans. Software Eng. vol SE-4, November 1978.

[21] R. W. Topor, "Interactive Program Verification Using Virtual Programs", Ph.D dissertation, Dept. Of Artificial Intelligence, University Of Edinburgh, Edinburgh, Scotland 1975.

[22] B. Wegbreit, " The Synthesis Of Loop Predicates", Commun. Ass. Comput. Mach, vol 17, February 1974.