3
Design of EDEN

The implementation of a definitive system is not a trivial task. As the name implies, EDEN was designed to ease the implementation of definitive systems. The language supports basic forms of definitions and features for defining operators.

The primary goals of the language are:

- to support general purpose programming (e.g. in order to implement definitive notations)
- to support definitions
- to support explicit actions (since they are extensible and much more powerful than the implicit actions, they fulfil the general purpose requirement)

3.1 Fundamental Philosophy

3.1.1 Formula Definition

A definition of an object is expressed as a mathematical expression. A variable is called a *formula variable* if its value is defined in terms of other variables, e.g:

$$\mathcal{F} \equiv \Phi(v_1, v_2, \ldots , v_n)$$

where $\mathcal{F}$ is the name of the formula variable, and Φ is a function of the variables $v_1, v_2, \ldots , v_n$. The variables $v_1, v_2, \ldots , v_n$ are called the *source* variables of this definition since the value of $\mathcal{F}$ is computed from their values.

The *source* $S(\mathcal{F})$ of a variable $\mathcal{F}$ is the set of variables on which the value of $\mathcal{F}$ depends. Suppose variables $v_1, v_2, \ldots , v_n$ appear on the R.H.S. of the definition of $\mathcal{F}$, i.e. $v_i \in S(\mathcal{F})$ for $1 \leq i \leq n$. Also, all variables belonging to $S(v_i)$ are in $S(\mathcal{F})$. Hence,

$$S(\mathcal{F}) = \{ v_1, v_2, \ldots, v_n \} \cup \bigcup_{i=1}^{n} S(v_i) \tag{3-1}$$

EDEN supports the concept of *formula*. A formula definition has the form:

identifier **is** *expression* ;

or

$$\mathcal{F} \textbf{ is } \Phi(v_1, v_2, \ldots, v_n); \tag{3-2}$$

The keyword "is" defines a formula variable $\mathcal{F}$ whose value is computed from the values of source variables $v_1, v_2, \ldots, v_n$ and the *expression* on the right hand side, denoted by Φ, is the formula of the variable $\mathcal{F}$. In other words, a formula describes how the value of a variable is computed from other data. These formulae are permanently valid (unless they are re-defined). That is, no matter what the values of source variables are, the value of variable $\mathcal{F}$ is *always* equal to $\Phi(v_1, v_2, \ldots, v_n)$. Thus a formula gives an abstract definition of a variable rather than the explicit value of it. This is the major difference between formula definitions and conventional assignment statements (denoted by the = operator). For example, after executing the assignment,

$$\mathcal{V} = \Phi(v_1, v_2, \ldots, v_n);$$

the value of $\mathcal{V}$ is equal to $\Phi(v_1, v_2, \ldots, v_n)$ only after the expression Φ is evaluated and before any of the values of the source variables is altered. In other words, the assignment operator takes a snapshot of the value of Φ at the instance of evaluation and stores the result in the variable $\mathcal{V}$.

Unlike the assignment statement, the formula definition, itself, is not an executable statement, i.e. the formula definition only assigns an expression to the variable; when and how a definition is evaluated is managed by the system. However, the values of the formula variables are guaranteed to be up-to-date when the variables are examined.

Consider the following definitions,

A is A; (3-3)
N is N + 1; (3-4)
F is G; **G is** F; (3-5)

All of these definitions are cyclically defined. That is,

$$\mathcal{F} \in S(\mathcal{F}) \tag{3-6}$$

Although (3-3) seems logically correct, the value of A is undefined since there is no way to determine what A is (unless there is a way to assign a value to it). In (3-4), it is obvious that N is logically unacceptable. The definitions in (3-5) are indirectly cyclically defined. These cyclic definitions are usually prohibited by the definitive systems; otherwise, the evaluation of these definitions may cause infinite loop.

Notice that a definition does not imply the inverse definitions (where these exist); for example, the definition (3-2) does not imply

$$v_i \text{ is } \Phi_i^{-1}(\mathcal{F}, v_1, v_2, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n); \qquad \text{where } 1 \le i \le n$$

A definition can be a representation of a real relationship among objects; for instance, "a lamp is placed (somewhere) on a table" can be formulated as:

```
lamp_position is table_position + something ;
```

If the table is re-positioned the lamp will also be moved because the definition has specified the position of the lamp in terms of the position of the table, the system can automatically keep track of these definitions and re-calculate `lamp_position` whenever `table_position` has been changed. But if the `lamp_position` is then redefined to

```
lamp_position is desk_position + something ;
```

we mean to put the lamp on the desk instead of the table. The table remains at the same position because it is not defined in terms of the lamp. Note that the new definition of `lamp_position` overwrites the old definition. Hence the lamp can, if we want, be moved independently by re-defining its formula. This is an important characteristic of definition.

3.1.2 Action Specification

A well-known example of a definitive system is the spreadsheet software [Am86]. A spreadsheet is a program that provides us with a matrix of cells where we can insert numbers and formulae. Each cell has a unique label composed of its coordinates; for example, R1C5 is the cell at row 1 and column 5. A cell that contains a formula causes the system to automatically recalculate the formula whenever any cell on which it depends is changed and display the formula's value. The location is determined by the coordinates

specified by its label. These display updating actions are built into the package. We may call these actions *implicit actions*.

The make utility in UNIX is a file maintenance program. User has to specify the file dependency in a *makefile*. The content of the target file is updated if the user has specified a UNIX command in the makefile. For example, to express the fact that

"object file file.o is compiled from C source file file.c"

The appropriate makefile is:

```
file.o : file.c
        cc -c file.c
```

The first line indicates the dependence and the second line contains a UNIX command showing how the object file can be compiled from the source file. The explicit actions allow less restricted updating actions to be defined. For example, the user can give different options to the compiler or use different compilers to compile the target object. However the user has to make sure there are no destructive side-effects.

Note that the command lines can be omitted since make understands what to do from the suffix of file names by some pre-defined rules. If the target's suffix and the source's suffix match a rule, the pre-defined action is performed. The users can override the rules by supplying their own actions or even define their own rules.

We may call these commands in the makefile *explicit actions* since they are defined explicitly by the user.

Generally an (implicit or explicit) action is a procedure (sequence of instructions) to perform some activities included updating the content of the target object, e.g. the lamp.position and file.o. In the case of explicit action, the update of the target object is optional. Because explicit actions allow a less restricted set of instructions, the applications can be more general. However the user has to take care of any unwanted side-effects.

Unlike spreadsheet software, EDEN has no default *implicit action* associated with each definition. This is because EDEN was not designed for a particular application. On the other hand, EDEN provides a way of defining explicit actions. Thus the user can specify special updating actions for different devices.

In the rest of this thesis, the term *"action"* refers to *"explicit action"* because it is the only kind of action available in EDEN.

The general form of an action specification in EDEN is:

proc *identifier* : *identifier-list* { *C-like-statements* }

The following statement illustrates a sample action definition.

```
proc display_v : v { writeln(v); }
```

The keyword **proc** defines an action, named as `display_v`, which is invoked by the system when the value of variable `v` (specified after the colon) is *changed* (the meaning of *change* shall be discussed later). The curly brackets { } enclose a list of statements to be executed sequentially. In this case, there is only one function call, the `writeln` function. By calling different functions (with appropriate side-effects, e.g. `writeln(...)` prints the values of its arguments on the standard output) in the function libraries, `display_v` can do different tasks, such as realize the data graphically. This makes EDEN more flexible.

Note that the value of `v` is not passed as a parameter to `display_v` when the action is invoked, but since it is a global variable, it can be accessed by the action directly.

3.1.3 Dependency Graph

We can use a dependency graph to study the relationships between variables. Each variable is represented by a node. A directed edge connects a variable to one of its direct sources — the variables appearing on the R.H.S. of its definition. Since the graph is acyclic, the nodes form a partial-ordered set, i.e. $V > W$ if V is dependent upon W.

An example of an informal description of a house and some other objects is given in Figure 3-1. The dependency graph of it is shown in Figure 3-2.

```
object: lamp, table, room, house

lamp :- base, cable, position, size, ...
table :- position, ...
room :- position, ...
house :- position,width, length, height, walls, ...
...
lamp.position ≡ table.position + ...
table.position ≡ room.position + ...
room.position ≡ house.position + ....
house.walls ≡ somefunction(house.width, house.length, house.height)
house.position ≡ (500, 500)                    #absolute coordinates
...
```

Figure 3-1: An informal description of some objects

The symbols ":-" and "≡" mean "is composed of" and "is defined as" respectively.

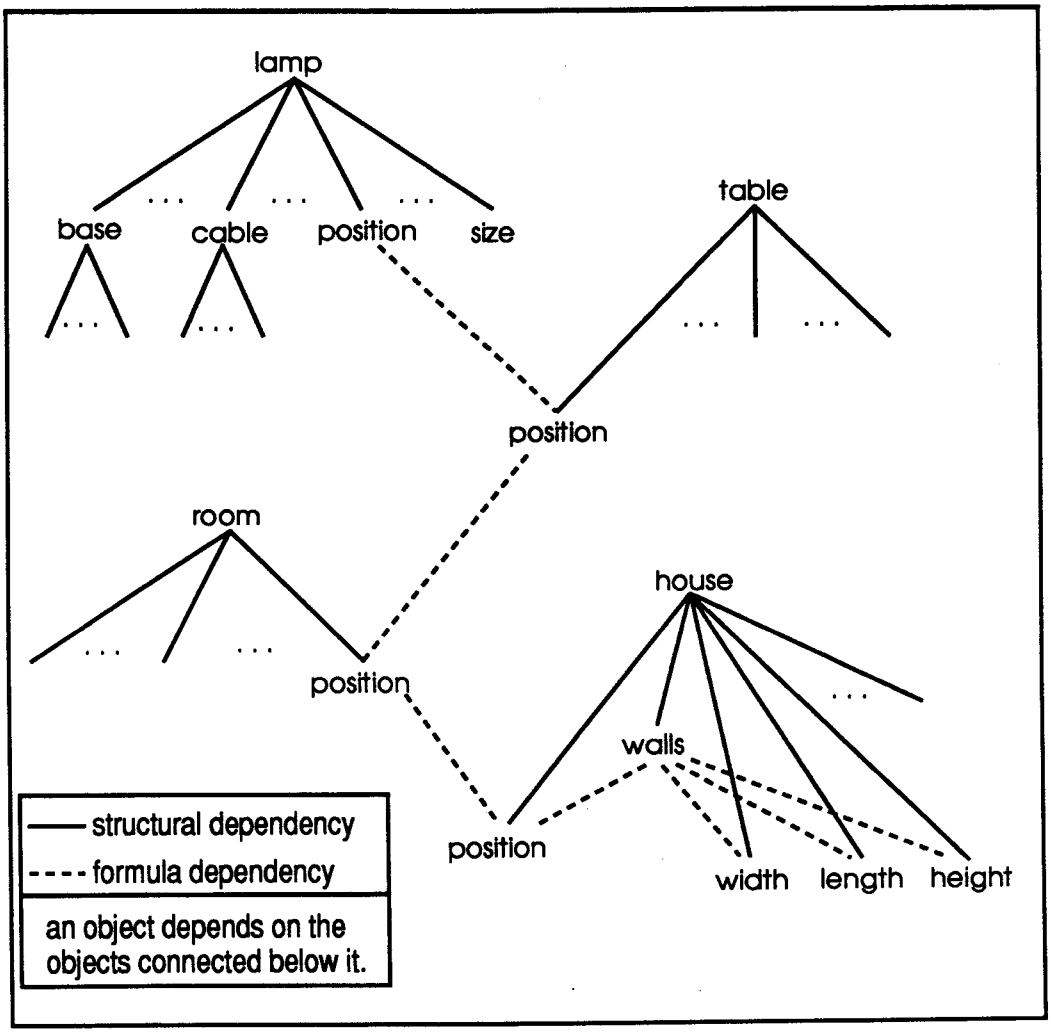

Figure 3-2 : The dependency graph of objects described in Figure 3-1.

Variables are denoted as nodes. The edges represent the data dependency. W is the direct source of V if there is an edge that connects V to W downwardly.

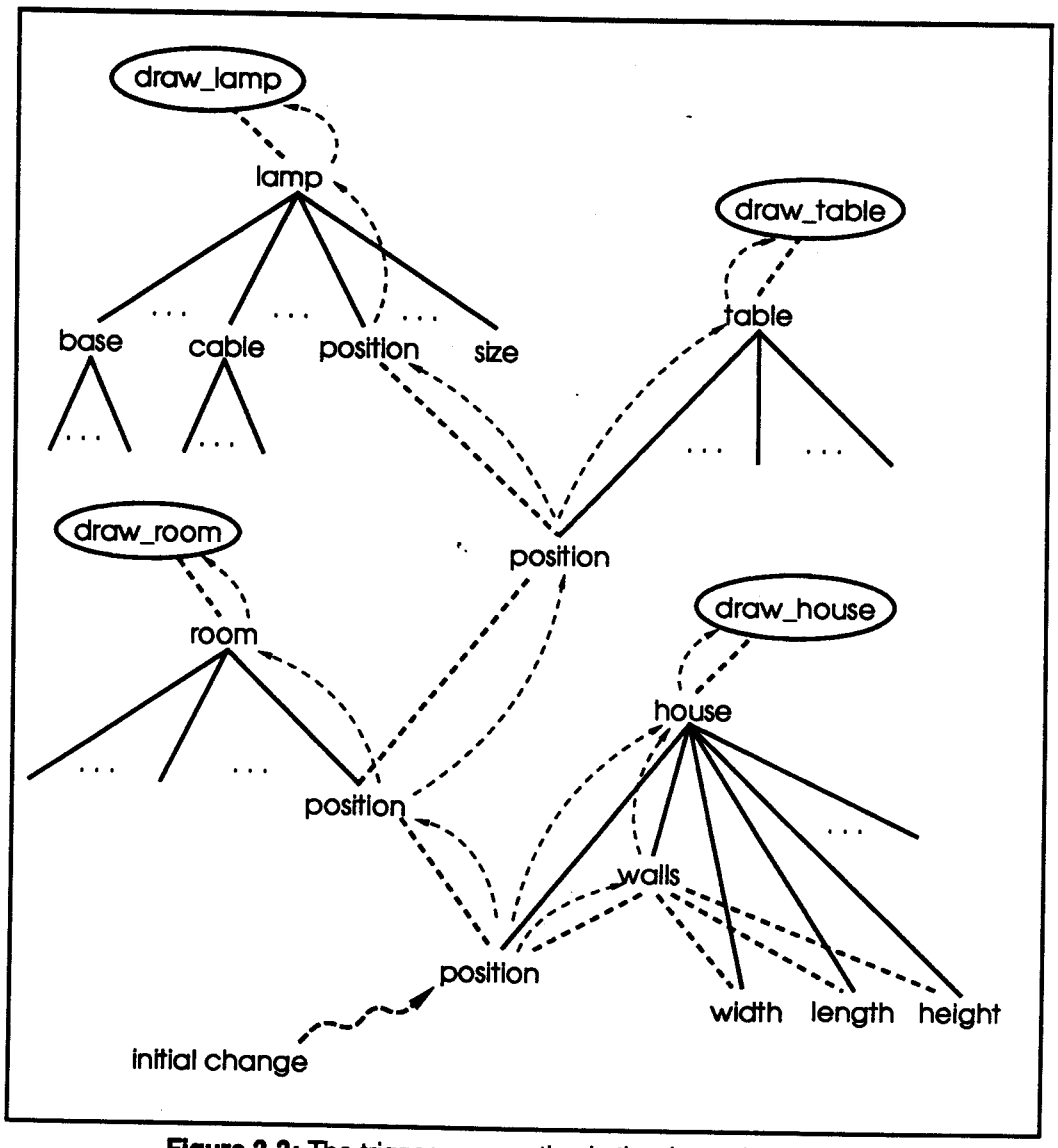

Figure 3-3: The trigger propagation in the dependency graph

Suppose the position of the house is changed. We can imagine that we trigger a switch of that node in the graph. The switch indicates whether the value of that node is up-to-date. This trigger causes the nodes connected above it to be switched too (like a chain reaction). The trigger propagates upwards through the graph.[1]

All the triggered nodes are then scheduled to be re-evaluated. Figure 3-3 shows the propagation. In the figure, actions (e.g. draw_lamp) associated to the

[1] The current implementation of the EDEN interpreter maintains a dependency graph. From the information of the graph, it can determine the order of evaluation of formulae and the order of execution of actions.

objects (e.g. lamp) are also shown. The trigger of lamp, for example, causes the associated action, draw_lamp to be scheduled.

3.1.4 Delay Evaluation and Execution of Definitions and Actions

In the current implementation of EDEN interpreter, evaluation of a definition is delayed until all values of source variables are **determined**. For example,

```
f is a + b;        /* f is not evaluated */
a is c + 1;        /* a is not evaluated */
c is 5;            /* c is determined and a is evaluated */
b is 6;            /* a and b are determined and f is evaluated
*/
```

Similarly, the execution of action is delayed until all the values of its source variables are determined. For example, in the last example, if we had defined the action:

```
proc echo_f : f { writeln(f); }
```

before f was defined, echo_f would not be executed until we did the last statement "b is 6", i.e. after f was evaluated.

However, we may force a definition to be evaluated. In this case, all undefined variables are assumed having the value @ (a special value means undefined). Some operators, such as arithmetic operators, accept this value as operands and return @, but the other operators might cause errors. For example, we executed the statement:

```
writeln(f);
```

after the definition of f and before the definition of a. "@" will be printed because both a and b are @, and @+@ gives @.

The fact that evaluation of variables is delayed until sufficient information is defined allows definitions to be introduced in any order.

3.1.5 Other Features

The *list* data structure is chosen to be the basic data type because:

- it is a dynamic data structure (it has an arbitrary number of elements with arbitrary types)

- it is easy to construct complex data structures
- it simplifies the language (list replaces other structured data types such as arrays)

Since *list* is a basic type, which consists of arbitrary typed elements, run-time type checking is necessary. Therefore, we need not declare a variable before it is used because the data is nevertheless checked at run-time. This leads to untyped variables though data are typed. That means a variable can hold data of any type; and the data could even change its type during its life time.

Since the EDEN language does not have structured data type variables, the structural dependency (see Figure 3-3) must be modelled directly by definitions. This point is illustrated in the implementation of DoNaLD (see chapter 6). For instance, the structural dependency of **openshape** S:

```
openshape S { point p, q }
```

can be translated into EDEN definition:

```
S is [ p, q ];
```

To be a general purpose language supporting definitions, EDEN allows the user to define their own functions using some C-like statements. The EDEN interpreter can be customized for different applications. In addition, an interface to other languages' routines is necessary so the language can be expanded easily. The C-like syntax and calling convention was adopted. The reason for choosing C-like style is:

- there are a large number of C routines available.
- since the EDEN interpreter is implemented in C, interfacing with C routines is easier.
- C-like syntax is familiar to the large number of C programmers.

3.2 The EDEN System

The EDEN language has definitions, actions and some conventional features. The conventional features are, for example, iterative loops, flow control statements, user-defined functions and read/write variables.

EDEN variables can be classified into 4 types: *read/write variables*, *functions*, *formula variables*, and *action specifications*.[2] EDEN has different statements to assign values to these variables.

3.2.1 Read/Write Variables

Read/write variables (RWV) are the same as the variables of conventional languages. As the name implies, the value of a RWV can be assigned and examined by the user. For instance,

```
A = 1;
```

will assign integer 1 to the RWV A. The value of a variable is referenced if the variable appears in an expression — e.g. the right-hand side of the assignment statement. For example,

```
B = A;
```

puts the value of A into RWV B.

For any RWV, $\mathcal{V}$, there is no source variable:

$$S(\mathcal{V}) = \varnothing$$

Note that, in EDEN, a function is considered as a RWV (although there is a different instruction to define a function, and some restrictions on their definition). Functions will be discussed in the later chapters.

3.2.2 Formula Variables

A *formula variable* (FV) consists of a *data register* (DR) and a *formula expression* (FE) (or simply *formula*). The formula describes how the value of the *data register* (DR) can be computed from other variables (RWV or FV). The system will always update the DR using the formula whenever it is possible. This part of the system is here called *formula data maintainer* (FDM). Figure 3-4 gives a block diagram of the *formula maintaining sub-system*.

---

[2] Note that, in EDEN, an action can be re-defined. Thus, action specifications can be regarded as variables.

The value of a formula is always obtained by referring the value of the DR of that FV. The user can (re-)define the formula of a definition but not the value of the DR. This means that the DR is read-only by the user. In EDEN, the formula of a formula variable is defined using the "is" operator, for example:

```
f is a + b ;
```

defines a FV `f` whose value is always equal to the sum of the values of variables `a` and `b`. The expression "`a + b`" is the FE of `f`. Note that all variables mentioned in the FE, in this case, `a` and `b`, can be either FV's or RWV's.

The value of a FV is referenced if the FV appears in an expression, e.g. the right-hand side of an assignment statement.

The syntax of a formula definition is given in the next chapter.

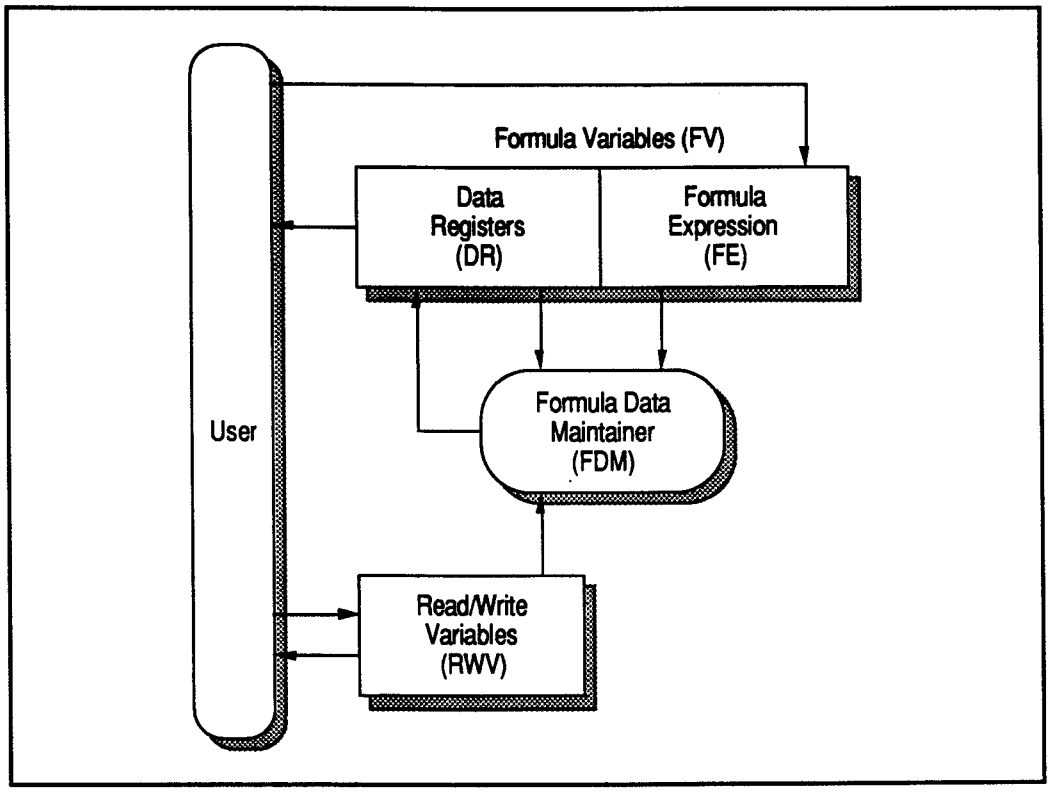

Figure 3-4: The block diagram of a formula maintaining sub-system.
The arrows show how data might flow. E.g. the user can define a FV by giving the formula to it. As shown in the diagram, the user can examine the value of a DR but not write to it. When the user (re)defines a FV or writes to a RWV, the FDM will re-calculate those DR's using their associated FE's. The FDM will read the values of some RWV's and/or DR's if it is necessary. The results will be assigned to the DR's by the FDM.

3.2.3 Action Specification

An *action specification* (AS) is a named sequence of instructions. This sequence of instructions will be invoked by the system whenever the values of any source variables, specified explicitly in a list, are *changed*. The term *"changed"* is causally defined. It may mean the value of a variable is different from the previous one, or the value of a variable is overwritten (by the user or by the system) though the value may be the same as the previous value. EDEN takes the latter definition. This definition of "changed" is used throughout this thesis unless otherwise specified.

An example of an action definition is:

```
proc print_sum : a, b, c
   { writeln("a+b+c=", a+b+c); }
```

This action, named as `print_sum`, will be invoked by the system whenever the values of the source variables `a`, `b` or `c` (listed after the colon) are changed.

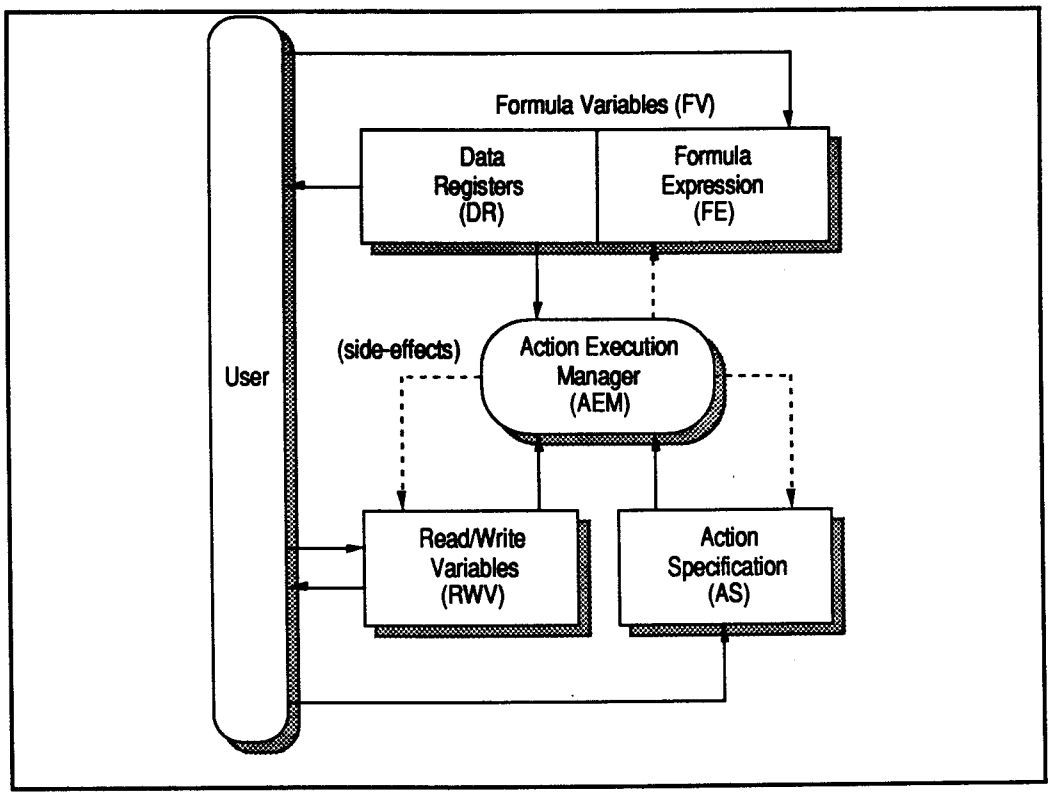

Figure 3-5: The block diagram of the action execution management sub-system.
The arrows indicate the possible data flow in the system. Dashed arrows indicate the data flow due to the side-effects of action execution. From this diagram, we see that the actions are invoked by the AEM. AEM loads the procedure of the appropriate action from the AS and then executes it. The execution may reassign the values of one or more RWV's, or (re)define FV's or AS's. Of course, the action may also access I/O devices (not shown in the diagram).

The action will print a string and the sum of the values of these variables on the screen due to the side-effect of the pre-defined function `writeln`.

The source of an action is defined to be the identifiers listed after the colon and before the action body. The above example gives

$$\mathcal{S}(\texttt{print\_sum}) = \{ \texttt{a}, \texttt{b}, \texttt{c} \} \cup \mathcal{S}(\texttt{a}) \cup \mathcal{S}(\texttt{b}) \cup \mathcal{S}(\texttt{c})$$

The *action execution manager* (AEM) is responsible for invoking the action. This sub-system is called the *action execution management sub-system*. Figure 3-5 gives a block diagram of it.

3.2.4 Function

Functions can be defined using the following syntax:

`func` *identifier* { *local-var-declaration*$_{opt}$ *statement-list*$_{opt}$ }

No formal parameters are needed in the function specification. The actual parameters always form a list called `$`. `$[`n`]` is the n^{th} argument. For convenience, `$[`n`]` can be replaced by `$`n, for n is an integer constant. The operator # returns the current number of items in a list. Hence, within the body of a function, `$#` is the current number of actual arguments.

Local variables are specified in *local-var-declaration*, preceded by the keyword **`auto`**. The *statement-list* is a sequence of C-like statements.

Example:

```
func max     /* returns the max value of its arguments */
{
   auto     i, m;                          /* local variables */
   m = $1;                                 /* the first argument */
   for (i=2; i <= $#; i++)
      if ($[i] > m) m = $[i];
   return m;                               /* returns the max */
}
```

Note that, **`proc`** is the same as **`func`** in EDEN. The keyword **`proc`** is intended to designate a procedure — a function whose value returned is meaningless or always be ignored.

To call a function, the actual arguments must be listed in parentheses preceded by the function name. The parentheses cannot be omitted even there are no arguments; otherwise, the function name denotes the function itself.

3.2.5 Data Types

EDEN has the following data types: @, integer, floating point, character, string, list, pointers. The integer, floating point and character types are very much like those in C. The pointers work like those in C except that no pointer arithmetic is available.

The constant `@` (meaning "undefined") has a unique (un-named) data type. All undefined variables are initialized to `@`. Some operators, such as arithmetic operators, accept `@` as their operands, and usually return `@`.

Strings can be thought of as an array of characters. Individual characters can be accessed by appending an index (an integer expression) enclosed in `[]`. The first character is numbered 1. The suffix operator, `#`, returns the number of characters in a string, and the infix operator, `//`, concatenates two strings. The pre-defined function, `substr`, returns a portion of a string. Comparison operators are extended to compare strings at the character level. The interpreter handles all memory management for string operations.

The pre-defined functions, `execute` and `include`, cause the interpreter to read a string as an EDEN program, and load a file (named by the string argument) as an EDEN program respectively. The function `execute` is useful for implementing macros, and `include` is useful for loading external EDEN codes.

The list is the only generic structured data type in EDEN. Data items of different types can be grouped to form a list using the list formation operator `[]`. Commas are used to separate the data items. The whole list is considered as a single data item. For example,

```
[ 100, 'a', "string", [1,2,3] ]
```

is a list holding four items: an integer, a character, a string and a list.

Individual data items can be accessed by giving an index (an integer expression) after the list. The first item is numbered 1. The suffix operator `#` returns the number of items in a list.

3.2.6 Operators

EDEN supports the following C-like operators:

- (negation)		** (indirect)*		*& (address)*
+	-	*	/	%
<	<=	==	!=	>= >
!	&&	||	?:	
=	+=	-=	++	--

Note that the assignment operators, =, +=, -=, ++, and --, cannot be used in definitions.

Non-C operators are:

#	//	[] *(list formation)*

3.2.7 Statements

EDEN supports the following C-like statements:

expression ;
if (*expression*) *statement*
if (*expression*) *statement* **else** *statement*
while (*expression*) *statement*
do *statement* **while** (*expression*) ;
for ($expr_{opt}$; $expr_{opt}$; $expr_{opt}$) *statement*
switch (*expression*) *statement*
case *constant* : *statement*
default : *statement*
break ;
continue ;
return $expression_{opt}$;
;
{ $statement\text{-}list_{opt}$ }

The followings are list-operation statements:

insert *list* , *position* , *value* ;

insert inserts an item having the *value* (of any type) to the *list* variable at the *position.*

append *list* , *value* ;

append appends an item having the *value* to the *list* variable (must be a RWV) at the end.

delete *list* , *position* ;

delete deletes an item from the *list* variable at the *position.*

```
shift list ;
```

shift deletes the first item from the *list* variable. If no *list* is specified, the argument list $ is assumed.

All *list* variables in these four statements must be RWV's.

With these list-operation statements, we are able to implement list-operators. For instance, the operators *head, tail* and *cons* can be written as:

```
func head                /* $1=list */
{
      return $1[1];      /* the head */
}

func tail                /* $1=list */
{
      shift $1;          /* remove the head */
      return $1;         /* return the tail */
}

func cons                /* $1=value, $2=list */
{
      insert $2, 1, $1;
      return $2;
}
```

3.3 Comparison of the Definition and Action Concepts

Although definition and action specification have syntactical differences, they share some common characteristics.

Differences:

- the source variables of a formula variable are implied in the formula expression; while those of an action are explicitly specified in the action specification.
- For a definition, after the evaluation, the result is stored in the data register. Since an action is a procedure, no value would be returned, and, therefore, no value has to be stored. (In fact, the value of an action is the pointer to the action procedure.)

Similarities:

- Both of them are triggered by the value change of source variables.

- Both of them are scheduled by the system to be evaluated or executed. Hence it minimizes the number of evaluations or executions. For example, consider

```
F is A + B;                                          (3-7)
```

and

```
proc action_F: A, B { value_F = A + B; }             (3-8)
```

if both the values of `A` and `B` are changed, `F` is evaluated once instead of twice. Similarly, `action_F` is executed once.

Despite the differences in syntax, definition and action can emulate the effect of each other.

3.3.1 Emulate Definition using Action

We can always express a definition in the form (3-2); we can always define an action, *formula_F*, such that,

proc *formula_F*: $v_1, v_2, \ldots, v_n$ { $\mathcal{F} = \Phi(v_1, v_2, \ldots, v_n)$; } (3-9)

For example,

```
right is left + width;                               (3-10)
```

can be emulated by

```
proc formula_right : left, width
      { right = left + width; }                      (3-11)
```

Note that `right` is not directly dependent on `left` and `width` in the emulation. The action is now responsible for maintaining the value of `right`.

3.3.2 Emulate Action using Definition

On the other hand, for any action, we can always define an equivalent action $\mathcal{A}$:

proc $\mathcal{A}$: $v_1, v_2, \ldots, v_n$ { *ProcBody* (); } (3-12)

where

func *ProcBody* { *original-procedure-body* }

The equivalent definition is:

action_A **is** *ProcBody* ($v_1, v_2, \ldots, v_n$) ; (3-13)

Notice that the arguments are always ignored by *ProcBody* since the action $\mathcal{A}$ will have no parameters when it is invoked by the interpreter. However, they cannot be omitted in the definition to establish the dependency.

For instance, the action specification:

```
proc print_V : V { writeln(V); }
```

can be replaced by

```
func echo_V { writeln(V); }
print_V is echo_V(V);
```

In this particular case, the program can be simplified to

```
print_V is writeln(V);
```

3.3.3 Discussion

Figure 3-5 shows the similarities of formula variables and action specifications at the machine level. The *data register/formula expression* pair is emulated by the *RWV/action specification* pair. The *action execution manager* replaces the *formula data maintainer* in Figure 3-4.

By comparing Figures 3-4 and 3-5, we can see the differences between them. The formula data maintainer normally updates the data register of the formula variable; while this "normal effect" of a definition is replaced by the "side effect" of the action procedure. In addition, the formula maintaining sub-system prevents the user writing to the data register in order to ensure the consistency of a definition but the action execution manager sub-system does not prevent the user writing to the RWV (data register counterpart) in the emulation.

Nevertheless, the emulation between definitions and actions is important because some restrictions imposed on definitions, such as non-cyclically defined restriction, can be circumvented by the equivalent action emulations. Consider the following **illegal** definition:

```
Sum is Sum + Data;                                    (3-14)
```

The original intention is to update the total sum by adding the new data to the current sum. (3-14) is illegal because `Sum` is cyclically defined and its initial value cannot be defined anyway. The equivalent action emulation (3-15) can solve the non-cyclic restriction by simply removing the dependency of `Sum`.

```
Sum = 0;
proc Total:Data { Sum = Sum + Data; }
```
(3-15)

We also add the line "`Sum = 0`" to initialize the `Sum`. However, even (3-15) is not a good solution to this problem. The difficulty lies in the algorithm of the solution itself. It is because the algorithm is a procedural one:

step 1: sum = 0
step 2: sum = sum + data
step 3: go to step 2

which could not be "well" expressed by definitions.

(3-15) is highly dependent on the triggering mechanism. The current implementation of EDEN causes a trigger if a value is assigned to a variable no matter whether the new value is different from the original value. However, if we change the triggering strategy to generate a trigger only if the new value is different from the original value in order to further optimize the evaluation, then (3-15) will fail; for instance, the data stream is:

2 3 5 5 4

the output sequence of `Sum` shall be:

0 2 5 10 14

rather than

0 2 5 10 15 19

On the other hand, suppose we change the definition of `Sum` to:

```
Sum is sum_of_items(DataList);
```

where `sum_of_items` is a function that returns the total sum of all items in the list `DataList`, and `Data` is now appended to `DataList` using, for example, the **append** statement. Then `Sum` will always generate the correct solutions.

Although the action is more flexible, the user has to specify the dependency explicitly. The definitions provide a better notation for specifying mathematical model, i.e. (3-10) is more meaningful than (3-11).

The definition-form actions illustrate the possibility of writing action specification in entirely definitive style.

However, we still want to distinguish action from definition for the primary reasons:

- The intention of action is different from that of definition. Action is mainly for display updating or similar effects. Definition is for modelling the problem.

- The actions help to isolate the I/O from the mathematical model. Once the action layer has been set-up, the user can concentrate on the definitions. The action layer functions independently reflect the current state of definitions.