4
EDEN Programming

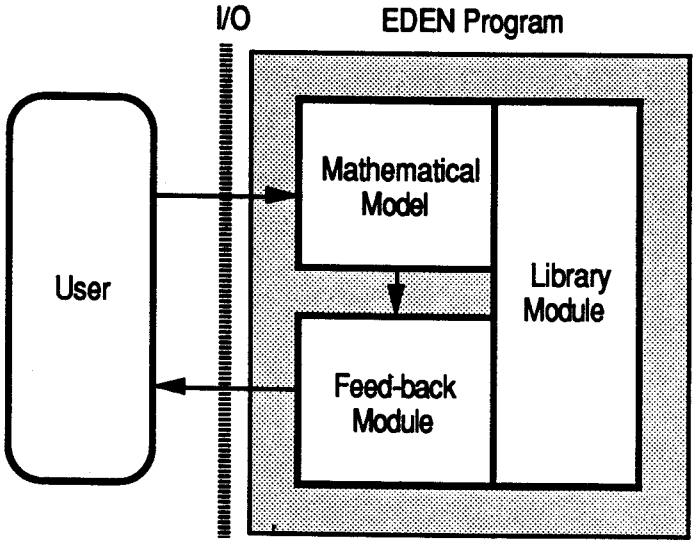
Maintaining computer programs is always a difficult task for the programmers. Modern languages, like **Modula-2** [TD89], have features to aid programmers to partition a program into small separable pieces—modules. A good modular design of program code is more easy to maintain and re-use.

A definitive program, especially the EDEN program, is not easily modularized. The reasons are:

- A definitive program is usually an interactive application where definitions can be modified by the user at run-time. It is very difficult to modularize such a volatile program.
- A definitive program is a set of definitions where each definition has certain dependences upon other definitions. Since definitions imply the inter-relationships among variables, variables can be difficult to modularize.

EDEN's imperative features, such as while-loops, can interact with the declarative definition dialect in complex ways and make the program hard to understand. In order to minimize these problems, EDEN users are advised to (at least mentally) partition an EDEN program into 3 modules:

1. **Mathematical Model**: a set of relationships (definitions) and parameters (read/write variables) that models the problem.
2. **Feed-back Module**: a set of actions that automatically reflect the up-to-date data to the user.
3. **Library Module**: a collection of functions and procedures that supports the other two modules.



While the program is running, the end-user keeps altering the parameters or definitions of the mathematical model. The actions provide a feed-back to the user to form an interaction loop. Usually, the details of the feed-back and library modules are hidden from the end-user.

The library module is a set of imperative functions/procedures which should be loaded before the other two declarative modules so that when the definitions/actions are evaluated/executed the functions/procedures are already defined.

4.1 A Window Specification Example

The motivation for studying definitive programming as a basis for software specification is similar to that behind the advocacy of functional programming methods, as cf. [GHT84]:

> *In recent years, extensive efforts have been expended in the development of rigorous methods for specifying, producing and verifying software and hardware products. They are far from successful with conventional computers and languages. The nature of the conventional approach presents difficulties since the notion of a global state that may change arbitrarily at each step of the computation has proved to be both intuitively and mathematically intractable. In other words it is very hard to guarantee the correctness and reliability of software.*

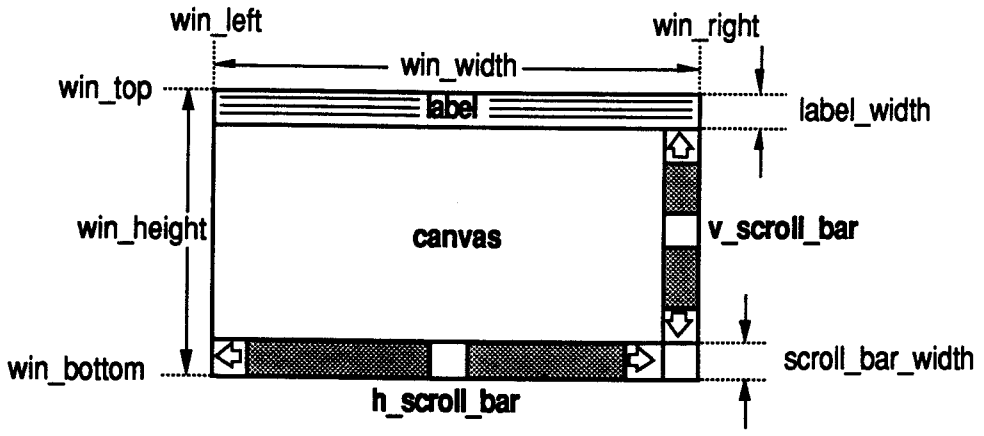
The major costs of software may be apportioned to the five parts of the software life cycle:

1. *Specification*
2. *Design*
3. *Implementation*
4. *Testing*
5. *Operation and Maintenance*

Although formal specification methods do exist, they are not widely used and there is usually a mismatch between the levels of language used in the specification, design and implementation phases. Thus the problem of deciding whether a problem meets the specification usually involves "exhaustive" testing and leads to software failure due to undetected errors, during the operation and maintenance phase.

The situation has been improved by the researches on the methodologies of formal specification [CHJ86; TM87] making the above quotation a little out of date. Definitive languages provide means of specifying computation using mathematical expressions to represent the relationships among data objects. In as much as the possibility of reasoning about relationships has yet to be explored, they are not as formal as many software specification languages c.f. [Su82; CHJ86; TM87]. The present status of definitive languages in software specification is discussed in [BRSYY89] and [BNRSYY89]. The interactive environments provided by definitive systems allow the user to design the mathematical model and test it when integrated into the system. Definitive systems, such as the EDEN interpreter, automatically and incrementally re-compile the specification (definitions) into machine code.

By way of illustration, consider various approaches to the specification of windowing software:



A window of a windowing system, such as **SunView** [SUN88a] or **NeWS** [SUN86; SUN87], may be described by the parameters: top, left, width, and height. The bottom and right can be computed from these parameters.

4.1.1 EDEN Style

In EDEN, the window can be described by the parameters and definitions (the mathematical model) as below:

```
...
label_width         =   16;
                                                        /*parameters */
scroll_bar_width    =   16;
win_top             =   100;
win_left            =   100;
win_width           =   500;
win_height          =   300;
...
win_right           is  win_left + win_width;                   /* definitions */
win_bottom          is  win_top + win_height;
canvas_left         is  win_left;                   /* window contains a canvas and scroll bars */
canvas_top          is  win_top – label_width;              /* pixels count from left to right */
canvas_right        is  win_right – scroll_bar_width;
canvas_bottom       is  win_bottom – scroll_bar_width;      /*  pixels count from top to bottom */
v_scroll_bar_top    is  canvas_top;
v_scroll_bar_left   is  win_right – scroll_bar_width;
...
```

The generation of the graphic image can be specified by the actions (the feed-back module) as:

```
...
proc draw_canvas : canvas_top, canvas_bottom, canvas_left, canvas_right
{
        ...
        draw_box(canvas_top, canvas_left, canvas_bottom, canvas_right);
        ...
}
proc draw_label : label_string, label_bottom, label_left, label_right
{
        ...
        draw_string(label_string, (label_left + label_right) div 2, label_bottom–1);
        ...
}
...
```

The library module defines the procedure draw_box:

```
...
proc draw_box                   /* parameters: top, left, bottom, right */
{
        moveto($1, $2);         /* top left */
        lineto($1, $4);         /* top right */
        lineto($3, $4);         /* bottom right */
        lineto($3, $2);         /* bottom left */
        lineto($1, $2);         /* top left */
}
proc draw_string { ... }
...
```

The user can re-position the window by simply performing the assignments:

```
win_top    =  20;
win_left   =  40;
```

The proof of the correctness of the mathematical model is straight-forward because the values of formula variables are ensured to be up-to-date by the system. Ensuring the correctness of the action specifications and function definitions is still as hard as in conventional languages. The problem is reduced to checking elementary functions without side-effects and simple procedural actions. (Nevertheless, EDEN is not designed for software specification; we just show the potential of correctness proof in definitive programming.)

4.1.2 Procedural Style

Since a procedural language, such as C, does not maintain the definitions, the user has to re-compute the right and bottom and then call the draw_canvas procedure:

```
win_top      =  20;
win_left     =  40;
win_right    =  win_left + win_width;          /* must re-evaluate these values */
win_bottom   =  win_top + win_height;
canvas_top   =  ...
...                                            /* then call the screen update procs */
draw_canvas(canvas_top, canvas_bottom, canvas_left, canvas_right);
...
```

4.1.3 Object-Oriented Style

In an object-oriented approach, a problem can be partitioned into manageable pieces by defining classes of objects that closely match the concepts of the application. The term *object class* is used to describe the abstract data type. An *object* is a variable that references an instance of a class.

Object oriented languages provide a better way of data hiding and data modularization. For example, the parameters of a window and all the window-related operators can be grouped to form a *window* class.

In an object oriented language, such as C++, the definitions of right and bottom can be simulated by modifying the assignment operator for top, left, width, and height, to re-compute the "definitions" and re-draw the window. An object class is created with top, left, and so forth as its data fields. This method is known as operator overloading [St86].

LOOPS, an object-oriented lisp [BS83], allows routines to be attached to data items. These routines can not only be explicitly called but also automatically executed whenever the associated data items are accessed or modified. Such use of *active values* is similar to the approach adopted in EDEN's actions.

Another way of maintaining formulae is to build a number of operators to set the value of each parameter and then do the appropriate updating.

In some object-oriented window systems, such as **NeWS**, a window belongs to an object class, e.g. LiteWindow. Each class may have some sub-classes, e.g. LiteWindow has a sub-class ScrollWindow. When a window is re-positioned, a message is sent to all its sub-objects. Each object can set up some message handlers to serve the messages. A message handler, when it receives a particular message, will perform the appropriate procedure. This procedure can also send messages to other objects or sub-objects, or even itself.

For example, the following **PostScript** [ADOBE85a; ADOBE85b] program is part of the source code of **NeWS'** LiteWindow class.

```
...
/stretchcorner {                    % - => - (Interactively stretch window via corner)
    ...
    ParentCanvas setcanvas
    ...
    BBoxFromUser /reshape self send          % send a message to itself
    ...
} def

/reshape { % x y w h => - (Re-shapes window from bounded box)
    /FrameHeight exch store     % store the new dimensions of
    /FrameWidth exch store      % frame canvas
    ...
                                % compute new client canvas width
                                % ClientWidth is FrameWidth – BorderRight – BorderLeft
    /ClientWidth FrameWidth BorderRight sub BorderLeft sub store
    ...
    ShapeFrameCanvas
    ShapeClientCanvas
    ...
} def
...
```

Part of method specification for *LiteWindow* class

The handler /stretchcorner sends a message /reshape to the window itself to reshape the window with the parameters (pushed on stack) determined by the procedure BBoxFromUser earlier.

In an object-oriented system, the maintenance of definitions has to be specified by the user rather than by the system as in a definitive system. For example, the

message handler /reshape has to re-compute the values of variables (c.f. definitions), such as /ClientWidth, and then re-draws the window by calling ShapeFrameCanvas, ShapeClientCanvas, etc.

These ways of maintaining definitions are not flexible. For instance, if we want to add a parameter to or delete a definition from the window specification, the whole set of operators in an object-oriented program has to be re-defined. It is because the definitions, e.g. /ClientWidth, are mixed with the procedural activities, e.g. ShapeClientCanvas. A message handling routine, a *method* in object-oriented programming terms, is like an action but it is *passively* activated by the host (who sends the message); while an action in EDEN is *actively* monitoring the source variables. This characteristic of actions makes them isolable from the mathematical model. Thus any modification of definitions and actions can be independently modified. Of course, an application programmer is unlikely to add a new parameter to a windowing system, but a system developer might.

4.1.4 Discussion

In this section, we emphasized the independence of definitions and actions. This characteristic of definitions and actions allows them to be modularized into separate modules: a mathematical model and feed-back module. The separation of definitions and procedural actions makes them easy to be modified and maintained.

4.2 Constraint Maintenance

Constraint languages are declarative languages that have been used in various applications such as simulation, modeling, and graphics. Constraint-based systems express relationships between objects in terms of constraints which are some equations that are permanently consistent. The constraint solver maintains the values of variables according to the constraints.

General-purpose constraint solvers usually employ equation solving techniques, such as linear programming. Examples are **Juno** [Ne85], **Ideal** [Va80; Va81]. Some languages use techniques based on theorem proving methods. For instance, the language **Bertrand** [Le87] is based on term rewriting method [Bu83].

However, the simplest and easiest to implement constraint satisfaction mechanism is called *local propagation of known states*, or just *local propagation* [SS79].

In this section, we try to apply definitive principles to the implementation of constraint solver that is based on the local propagation technique. This is natural since a definition has a similar property to the constraint of a constraint-based language — they both maintain a relationship among variables. However a definition is "one-way" relationship, i.e.

$$v_1 \textbf{ is } f_1(v_2, v_3, v_4, \ldots, v_n);$$

does **not** imply

$$v_2 \textbf{ is } f_2(v_1, v_3, v_4, \ldots, v_n);$$

The *unidirectional constraint* described in [SM88] is more or less the same as a definition.

4.2.1 Method I — "terminated by comparison"

We might expect one (or more) definitions to do the same job as constraint does. For instance, the constraint "a≡b" might be represented by the pair of definitions "a=b" and "b=a". However, such a pair of formulae is not allowed by most of the definitive systems (EDEN is one of them) since it introduces cyclic reference. However, as explained in §3.3.1, a definition can be replaced by a pair of RWV and an action where the dependency becomes indirect, thus solving the acyclic restriction imposed by the definition.

Consider the simple constraint — the Celsius-Fahrenheit convertor — containing two variables, C and F:

$$F \equiv C \times 1.8 + 32$$

where F and C are the temperature in Fahrenheit and Celsius respectively. This constraint can be decomposed into a set of actions (two actions in this case):

on C change:	$F \leftarrow C \times 1.8 + 32$	(4-1)
on F change:	$C \leftarrow (F - 32) / 1.8$	(4-2)

The equivalent EDEN transformation of actions (4-1) and (4-2), named as `actionC` and `actionF` respectively, is:

```
proc actionC: C { F = C * 1.8 + 32; }
proc actionF: F { C = (F - 32) / 1.8; }
```

However these two actions cause livelock, for example, once C is given a value, the actionC will change the value of F and cause actionF to be executed. ActionF will then recalculate C and activate actionC — an infinite loop forms.

On the other hand the values of variables will be stable (unchanged) when the constraint holds; hence the loop can be terminated. Therefore the new value of the variable, say C, is stored in a temporary variable, say X, rather than stored in C. If X and C are different, the value of X will be put into C; otherwise, nothing will be done in order to terminate the recursion. We may call this scheme "*terminated by comparison*". The complete program of the Celsius-Fahrenheit convertor is shown in Listing 4-1.

Listing 4-1:
A simple Celsius-Fahrenheit constraint (terminated by comparison)

The keyword **auto** *declares local variables in procedures. The* **if** *statements check the terminate condition before assigning value to the constraint variables.*

```
proc actionC: C
{
   auto X;

   X = C * 1.8 + 32;
   if (F != X) F = X;
}

proc actionF: F
{
   auto X;

   X = (F - 32) / 1.8;
   if (C != X) C = X;
}
```

To test this program, the following was input:

```
C = 100; writeln("C=",C); writeln("F=",F);
```

The output was

```
C=100
F=212.000000
```

Then input

```
F = 0; writeln("C=",C); writeln("F=",F);
```

The output follows

```
C=-17.777778
F=0.000000
```

(4-3)

Weaknesses:

In Listing 4-1 the mathematical property is assumed:

$$f^{-1} \cdot f = f \cdot f^{-1} = id \text{ } \textit{(identity function)}$$

i.e.

$$f^{-1}(f(x)) = f(f^{-1}(x)) = x$$

Unfortunately due to the numerical errors, such as rounding error, the property cannot be computed exactly correctly. This explains why the output (4-3) in the test was "`F=0.000000`" (a floating point) instead of "`F=0`" (an integer, as the input) — one more iteration happened since "`C*1.8+32`" had a small rounding error. Luckily this error is negligible in the next step "`(F-32)/1.8`" giving the same result "`C=-17.777778`" and hence terminating the loop. Nonetheless, Listing 3-1 has a possibility of livelock in some circumstances. To avoid it, a small rounding error should be allowed in the termination condition checking, i.e.,

`(C != X)` and `(F != X)`

should be replaced by

`(|C - X| < ε)` and `(|F - X| < ε)`

respectively.

Such a solution introduces numeric errors and, even worse, an appropriate value of ε is difficult to determine in general — that makes this solution not very practical.

4.2.2 Method II — "terminated by status"

Another solution to get around this problem is to use a status flag to control the loop. The flag records the status of constraint — whether the constraint is activated externally (by the program outside the constraint) or internally (by the constraint satisfaction process). Let's call this status flag `S`. The program in Listing 4-1 is modified to Listing 4-2. We may call this scheme "*terminated by status*".

Listing 4-2:
A Celsius-Fahrenheit constraint (terminated by status)

`S==0` *if activated externally.* `S==1` *if activated internally. Thus* `S` *must be initialized to 0.*

```
S = 0;   /* initial */

proc actionC: C
{
   if (S == 1)
      S = 0;    /* stop */
   else {
      S = 1;
      F = C * 1.8 + 32;
   }
}

proc actionF: F
{
   if (S == 1)
      S = 0;    /* stop */
   else {
      S = 1;
      C = (F - 32) / 1.8;
   }
}
```

Notice that when `S==0` (activated externally), the flag `S` must be set to 1 before re-calculating `C` or `F` because the order of evaluation and execution of actions preformed by the interpreter are unknown. After the assignment has finished, the other action activates. Since `S` is now 1, `S` is reset to 0 and stop. The same output is returned in the test except (4-3) now becomes "`F=0`".

In the general case, a constraint with n variables, V_i, $1 \le i \le n$, there are n actions ($action_{V_i}$, $1 \le i \le n$). The general form of the actions (where each action affects one variable only) is shown in Listing 4-3.

Listing 4-3:
General form of constraint actions where each action affects one variable only

$action_{V_i}$ *depends on* V_i *and affects* V_j, *where* $j \neq i$.

```
proc action_Vi: Vi
{
   if (S == 1)
      S = 0;
   else {
      S = 1;
      Vj = f_Vi(V1,...,Vn);
   }
}
```

This scheme works because, for every variable, V_i, there is one and only one action, $action_{V_i}$. A change on V_i activates $action_{V_i}$, which will then activate $action_{V_j}$ ($j \neq i$) with S set to 1. $Action_{V_j}$ will reset S to 0 and terminate.

The remaining problem of this scheme is that the user has to define n functions, f_{V_i}, one for each action, $action_{V_i}$. In the next section, we try to overcome this problem by introducing constraint operator templates.

4.2.3 Method III — "constraint operator templates"

In the last section, the constraint is maintained by a set of actions. However, the functions f_{V_i} must be defined explicitly by the user. In this section, basic *constraint operators* are introduced to construct complicated constraints. A *constraint operator* is a simple operator which maintains a basic constraint relation such as addition and multiplication. Complicated constraints can be constructed by connecting constraint operators and variables by some *connector operators*. For the same Celsius-Fahrenheit Convertor example:

$$F \equiv C \times 1.8 + 32$$

The operators, "×" and "+", are now constraint operators. They are both binary operators consisting of two inputs (namely a and b) and one output (r). Figure 4-1 shows the abstract diagram of these operators.

Constraint Operator

Consider the "+" operator, the constraint of it is:

$$r \equiv a + b \tag{4-4}$$

The three possible actions are, for example:

$$\begin{aligned} &\text{on a:} \quad r \leftarrow a + b \\ &\text{on b:} \quad r \leftarrow a + b \\ &\text{on r:} \quad b \leftarrow r - a \end{aligned} \tag{4-5}$$

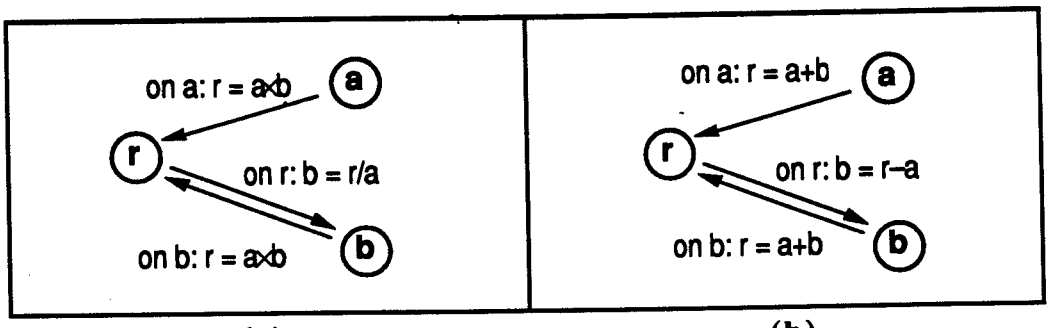
Since they are all symmetric, they have the general form:

$$\text{on } V_i\text{: } V_j \leftarrow f_{V_i}(a, b, r)$$

where

$$V_i \neq V_j \in \{a, b, r\}$$

The EDEN code of them can be generated by macros using the general code listed in Listing 4-3 as the basic model.



(a) (b)

Figure 4-1 Abstract diagram of **(a)** "×" and **(b)** "+" constraint operator:.
The sub-variables a *of both constraint operator templates are read-only within the operator's scope.*

Note that there are many possible ways to maintain a constraint relation. For example, we can have the action:

$$\text{on r:} \quad a \leftarrow r - b$$

instead of (4-5) while the constraint (4-4) can still be satisfied. The actual constraint satisfier may select different actions according to the adjacent constraint templates. However since we only want to demonstrate how easily we can implement constraint templates in EDEN, we do not consider these complicated cases. ([Le87] discusses the local propagation method.)

Connector Operator

A connector operator is a pair of constraint actions which copy the value of one variable to another variable and vice versa but mutually exclusive. Figure 4-2 shows the abstract diagram of a connector operator. Since the actions consist of copy (assignment) statements only (i.e. no computation errors), the "terminated by comparison" scheme in Listing 4-1 can be used instead of the "terminated by status" scheme as in Listing 4-3. The "terminated by comparison" scheme

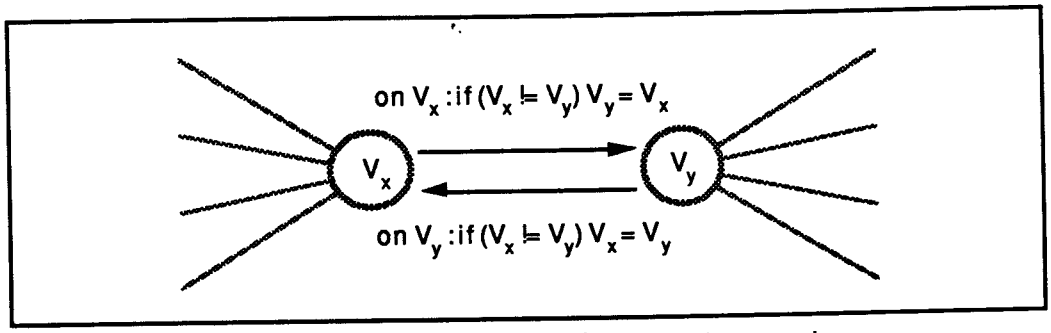


Figure 4-2: Abstract diagram of a connector operator.
Note that the two actions are the only components of the connector. The shaded parts, including the variables V_x *and* V_y*, do not belong to the connector.*

involves no extra variables (the status flag S), and the code for it is simple (see Listing 4-4).

Listing 4-4:
A sample program of a connector operator

```
proc action_x : V_x
{
   if (V_x != V_y) V_y = V_x;
}

proc action_y : V_y
{
   if (V_x != V_y) V_x = V_y;
}
```

The condition test (V_x != V_y) *can be used because the actions involve no arithmetic operations, i.e. no rounding errors.*

Figure 4-3 shows the abstract diagram of this Celsius-Fahrenheit convertor. The complete program is listed in Appendix B. The main program is very simple (only 7 function calls) and listed in Listing 4-5. The first two `new_op()` construct the ADD and MULT operators; the constraint actions are given in the second argument. The variables, named `ADD` and `MULT`, are created by `new_op()` with the list of name of constraint variables stored in them; i.e., `ADD` and `MULT` store the lists `["ADD_a","ADD_b","ADD_r"]` and `["MULT_a","MULT_b","MULT_r"]` respectively. For expressiveness, variables `a`, `b`, and `r` are assigned with `1`, `2`, and `3` respectively for indexing these strings.

The two `set()` set the sub-variable a of ADD and MULT, named by the strings `ADD[a]` (`"ADD_a"`) and `MULT[a]` (`"MULT_a"`), to some values given in the second arguments.

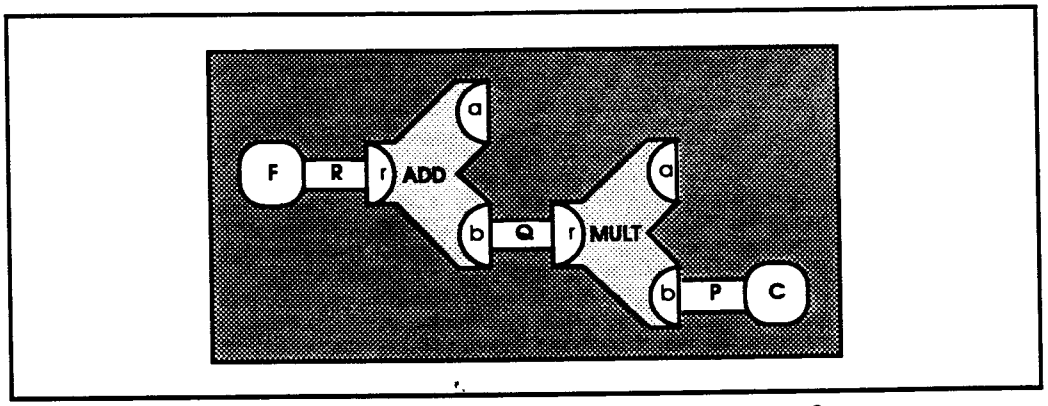


Figure 4-3: Abstract diagram of the Celsius-Fahrenheit Convertor constructed using constraint operators.

ADD *and* MULT *are the "+" and "×" operator respectively.* P, Q *and* R *are connector operators. Since both sub-variables* a *of* ADD *and* MULT *are read-only, they can be set to the constants* 32 *and* 1.8 *respectively.*

Finally, the last three `connect()` connect the variables (named by second and third arguments) by the connector named by the first argument.

As expected, the same result is given in the test as in §4.2.2.

Listing 4-5:
Main Program of the Celsius-Fahrenheit Convertor constructed by basic constraint templates

```
new_op("ADD",   [ r_is_a_plus_b,
                  r_is_a_plus_b,
                  b_is_r_minus_a
                ]
      );

new_op("MULT", [ r_is_a_times_b,
                  r_is_a_times_b,
                  b_is_r_div_a
                ]
      );

set(ADD[a],  32 );
set(MULT[a], 1.8);

connect("P", MULT[b], "C"     );
connect("Q", ADD[b],  MULT[r]);
connect("R", "F",     ADD[r] );
```

`new_op()` declares a new constraint operator named by 1st argument and the action functions are given in the 2nd argument. `set()` sets the variable named by the 1st argument to the value given in the 2nd argument. `connect()` connects two variables named by the 2nd and 3rd arguments; the connector is named by the 1st argument. (More detail in Appendix B)

The procedures `new_op()` and `connect()` hide the detail of variables and action generation. Internally, the function `macro()` (see Appendix B) is used to simulate the effect of macros substitution with the cooperation of the built-in function `execute()` (see Appendix A). This macros technique is used extensively to duplicate similar codes.

4.2.4 Discussion

In this section, constraints are expressed as a set of mutually exclusive definitions which are simulated by a set of actions. For a particular constraint, we can give a short EDEN program to maintain it. However the user has to define constraint actions for different constraints explicitly.

§4.2.3 is a more ambitious attempt to solve the general constraint maintenance problem by introducing basic constraint templates. Complex constraints can be constructed by these basic constraints. This approach sounds promising if we can build a constraint template library. However the shortcomings (lacks of structural data types) of EDEN prevent future investigation (although this problem can be solved by using macros, the EDEN program becomes confusing to the reader).

The approach of §4.2.3 is similar to the *local propagation* technique used by some constraint-based systems. The main differences are in the *firing* mechanism. Like the local propagation technique, some constraints, like:

$$\text{square\_x} \equiv x \cdot x$$

cannot be solved by the described method. Solving these constraints requires some general purpose equation solvers which are based on numeric methods, like linear programming, or theorem proving techniques. These alternative techniques are more difficult to implement and slower than the local propagation technique. Some discussions on different constraint solving techniques can be found in [Le87].

4.3 Simulation of Parallel Data Flow Machine

In this section, we try to define a data flow machine using formulae and actions. We consider the 3rd order polynomial:

$$f(x, a, b, c, d) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$$

If only binary data flow templates are used, a data flow machine can be built as shown in Figure 4-4. Each data flow template simulates an individual processor. The processors are connected by wires. There are 5 multiplication processors, 3 addition processors, and 3 delayers. Each processor is clocked by a common clock signal (not shown in the diagram). The data flow network forms a 4-stage data-pipeline. The purpose of the delayers is to hold the data for the later stage. The internal structure of a processor is shown in Figure 4-5. Each processor consists of three latches L1, L2 and L3 which are clocked by the line clock. L1 and L2 latch data from processor inputs A and B to the ALU inputs a and b respectively on the negative-edge of clock. L3 latches data from ALU output r to processor output R on the positive-edge of clock. The function of ALU is controlled by the line F.

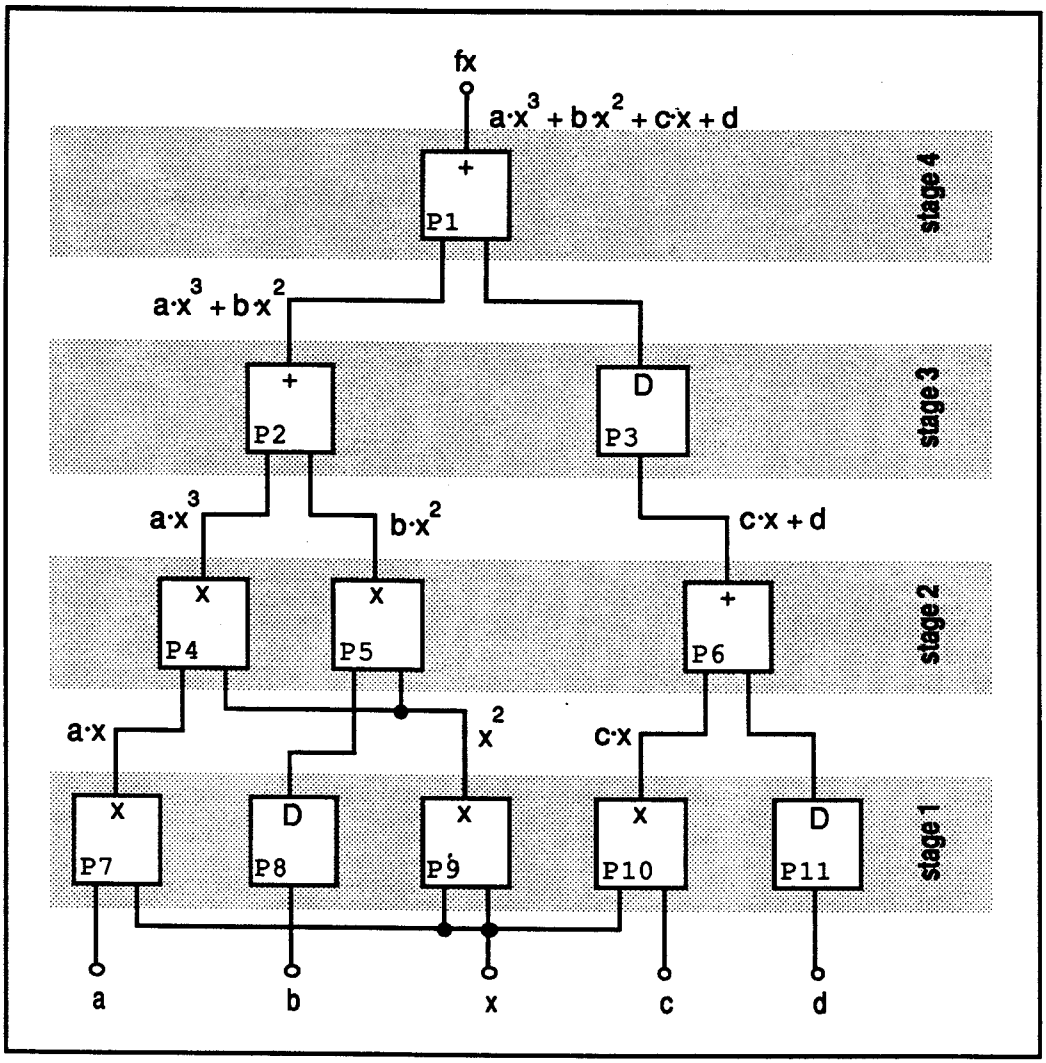


Figure 4-4: A pipeline arrangement of processors for evaluating the polynomial:
$f(x,a,b,c,d) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$.

Data flows upwards. The clock *is not shown in the diagram. The function of the processors are indicated by the symbols* + *(addition),* x *(multiplication), and* D *(delay). The processors are identified by the label at the bottom left hand corners of the processor templates.*

The processors are interconnected to form a pipeline with 5 input lines x, a, b, c, and d whose data is come from 5 data streams. The pipeline is expected to generate an output data stream at fx.

The following assumptions are made:

A1 Data flows in one direction (upwards in the diagram); i.e. no feed-back.

A2 Wires transfer data in one direction only (by A1) without any delay.

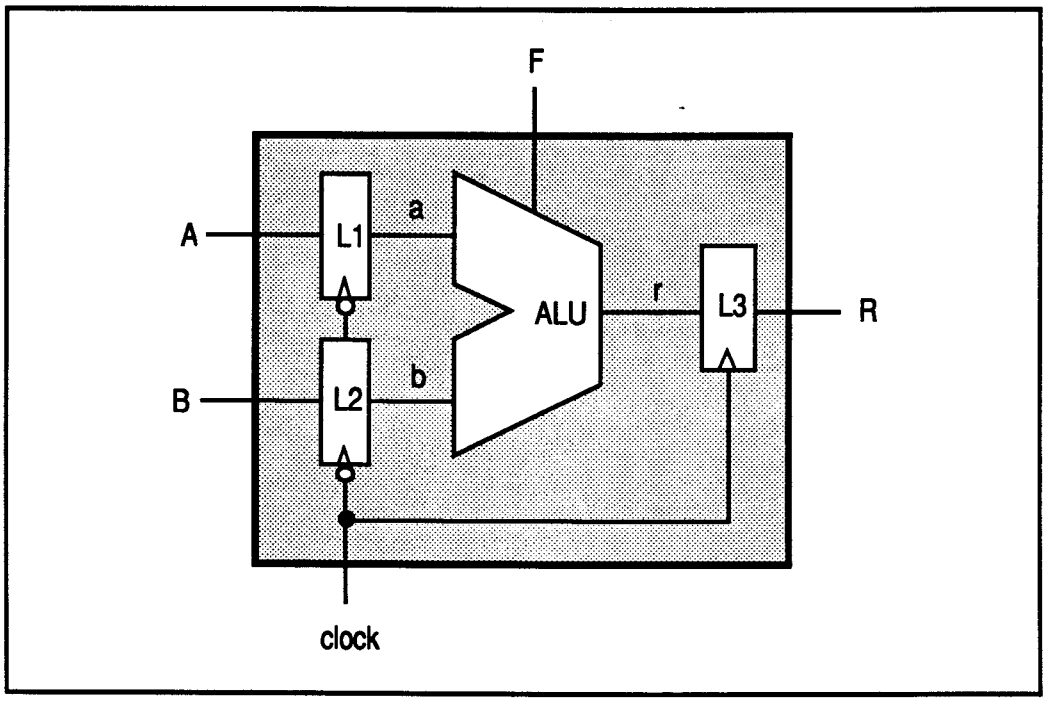


Figure 4-5: Diagram of a Processor Template.

A3 Processors are synchronized by the signal clock. The clock is oscillated between two possible states, either 0 or 1.

A4 All computation (addition, multiplication, or delay) of the processors consumes the same amount of time, i.e. one clock cycle.

A5 Inputs are loaded before clock = 0.

By A2, each inter-processor connection wire can be formulated by a single formula:

destination **is** *source*;

The latches inside a processor can be programmed as:

```
proc L1: clock { if (clock==0) a = A; }
proc L2: clock { if (clock==0) b = B; }
proc L3: clock { if (clock==1) R = r; }
```

The actions are activated when the clock is changed (i.e. on the edges of the clock). If the action finds that the clock is 0, it must be on the negative edge; otherwise it must be on the positive edge (by A3).

The ALU is simply

```
r is F(a, b);
```

where F is the operation function and can be set to one of the three functions: ADD, MULT, or DELAY.

Finally, the clock is oscillated externally by a while loop, and the inputs x, a, b, c, and d are loaded (from 5 different streams) before clock = 0 (by A5).

Discussion:

In this simulation program, a set of definitions was given by the user to describe the behaviour of each processor and their connectivities. To describe the problem, an object-oriented approach had been taken: the basic elements of a pipeline are processors and inter-processor wires. The processors can be decomposed into some latches, ALU and some intra-processor connection wires (a, b and r).

Then the properties of objects and the relationships between them and their neighbouring objects are carefully analyzed. For example:

- The latch L1 has 3 neighbours: clock, a and A; L1 operates when clock goes to 0 and its operation is to set a to A.
- The ALU has 4 neighbours: a, b, r and F. The ALU is an asynchronized device, and the end result r is **always** equal to F(a,b). (The time for computing the answer is assumed to be irrelevant.)

These relationships can then be translated to EDEN actions and formulae. Some procedural activities, such as triggering the clock, can be defined as procedures. The complete program is listed in Listing 4-6. (The macro expanded program is listed in Appendix C)

Listing 4-6: An EDEN program of a data pipeline of solving the polynomial: $f(x, a, b, c, d) = a \cdot x^3 + b \cdot x^2 + c \cdot x + d$. **(a)** Main program of definitions of the processor templates. **(b)** Code for testing the definitions. **(c)** Test output.

Listing 4-6(a): MAIN PROGRAM

```
include("macro.e");

                    /*—— ALU FUNCTIONS ——*/
func ADD    { return $1 + $2; }   /* r=a+b */
func MULT   { return $1 * $2; }   /* r=a*b */
func DELAY  { return $1;      }   /* r = a */

proc alloc_processor     /* $1=processor id */
{
   execute(macro(
     "proc ?1_L1: clock { if (clock==0) ?1_a = ?1_A; }
      proc ?1_L2: clock { if (clock==0) ?1_b = ?1_B; }
      proc ?1_L3: clock { if (clock==1) ?1_R = ?1_r; }
      ?1_r is ?1_F(?1_a, ?1_b);
     ", $1
   ));
}

                /*—— ALLOCATE 11 PROCESSORS ——*/
for (p = 1; p <= 11; p++)
   alloc_processor("P"//str(p));

                /*—— PROGRAM THE PROCESSORS ——*/
P1_F = P2_F = P6_F = ADD;
P3_F = P8_F = P11_F = DELAY;
P4_F = P5_F = P7_F = P9_F = P10_F = MULT;

                  /*—— INTER-PROCESSOR WIRES ——*/
fx     is P1_R;
P1_A   is P2_R;      P1_B   is P3_R;
P2_A   is P4_R;      P2_B   is P5_R;
P3_A   is P6_R;
P4_A   is P7_R;      P4_B   is P9_R;
P5_A   is P8_R;      P5_B   is P9_R;
P6_A   is P10_R;     P6_B   is P11_R;
P7_A   is a;         P7_B   is x;
P8_A   is b;
P9_A   is x;         P9_B   is x;
P10_A  is x;         P10_B  is c;
P11_A  is d;
```

Listing 4-6(b): TEST

```
x_stream = [ 2, 3, 4, 5, @, @, @, @]; /*—— INPUT STREAMS ——*/
a_stream = [ 1, 1,-4, 2, @, @, @, @];
b_stream = [ 3,-3, 5, 6, @, @, @, @];
c_stream = [ 3,-3, 5,-2, @, @, @, @];
d_stream = [ 1, 1, 2, 7, @, @, @, @];

for (time = 1; time <= x_stream#; time++) {
   write("time = ", time);
   x = x_stream[time];        /* load inputs */
   a = a_stream[time];        /*   before clock=0 */
   b = b_stream[time];
   c = c_stream[time];
   d = d_stream[time];
   clock = 0;                 /* trigger clock */
   clock = 1;
   writeln(", fx = ",fx);     /* print result */
}
```

Listing 4-6(c): TEST OUTPUT

```
time = 1, fx = @
time = 2, fx = @
time = 3, fx = @
time = 4, fx = 27
time = 5, fx = -8
time = 6, fx = -154
time = 7, fx = 397
time = 8, fx = @
```

In this example, we illustrate how a digital circuit can be specified using definitions. We assumed that each operation finishes in one clock cycle; however, we can model multi-cycle operators as pipelines of single-cycle operators. Similarly, we can also approximate analog components by modelling them as pipelines of digital operators.