Chapter 4

The Engine for Definitive Notations, EDEN

The EDEN tool is "An Engine for Definitive Notations" [Yun90]. In contrast to `am` and the DAM machine, which use the evaluation/storage strategies 1 ("evaluate-at-use") and 2 ("evaluate-at-redefinition") respectively, EDEN uses the hybrid evaluation/storage strategy 3, "evaluate-at-use-when-out-of-date" (see §2.2.1).

This chapter gives an overview of the evolution of the EDEN tool, an evolution which has spanned 17 years at the time of writing. The history is provided in order to place more recent work by the author and others into context. In order to compare EDEN to the ADM and the DAM machine, some of the low-level details of the implementation of the EDEN tool are described and analysed.

The EDEN tool implements the Eden language — in this thesis, I differentiate the two through use of capitalisation. The Eden language (documented in [Yun89]) uses a C-like syntax, providing a conventional procedural scripting language including the procedural assignment operator '`=`'. Unusually, it also provides a way to make (re)definitions, using the definitive '`is`' construct. The EDEN tool has played a major part in the practice of the EM group and Eden has provided a language for discussion of concepts over many years. The EDEN tool has a variety of front-ends and graphical facilities through the provision of other definitive notations, for example, DoNaLD, a definitive notation for line drawing, and SCOUT, a definitive notation for screen window layout. The various notations are implemented by

translation into the Eden language which is then executed by EDEN.

Eden is thus a procedural-definitive hybrid, allowing both dependency and agency to be programmed or modelled. In this respect, it is more general than the DAM machine, in which the focus is upon dependency. Compared to the ADM, Eden offers a more conventional model. Whilst the ADM could be used in a purely procedural manner by simulating control flow through the use of guards, this would be totally out of keeping with the spirit of intended ADM use.

Automated agency is possible in Eden through the use of a 'triggered action' facility. This was initially developed in order to link external graphical state to internal definitive state. As described in §4.3.7, one of the contributions of my work on EDEN has been to discover that the facility can be used to create definitive state definitively.

Automated agency is generally not considered in the DAM machine. Regarding the use of such agency to link external state to internal, consider Allderidge's attempts to add side effect to operators in order to link external graphics to internal state in §3.3.4. Regarding the use of agency to link state internally, DAM machine operators could invoke `addtoq` in order to modify internal state, but the re-entrant nature of the DAM machine is not well understood (see §3.2.6). The DAM machine also schedules the entire update in phase 2 of the BRA before it is actually performed in step 3 (see Figure 3.3 on p.118). Changes to the script graph as operators are invoked will therefore have no effect on the current `update` process. In contrast, as described in §4.3, EDEN constructs a schedule in a more flexible manner, taking account only of adjacent targets of the 'current' node when scheduling change.

Compared to the guarded command lists of the ADM, Eden's actions are relatively primitive. At some level of abstraction, each Eden action is an 'atomic' sequence composed of smaller actions. Eden actions are not interleaved in execution: each runs to completion before the EDEN scheduler determines which to execute next. The invalid transition is not a concept that applies in this sequential scenario, and EDEN makes no checks for conflict between actions. EDEN also lacks two other major features of the (albeit as yet inadequately implemented and little used) ADM:

- entities are not represented — as a result, instantiation and removal of blocks of definitions involves parsing the symbol table as a set of strings, and
- there is no manual control 'debugger-like' stepping mode.

This chapter is organised as three main sections. Section §4.1 discusses the development of EDEN from its first conception by Y.W. Yung and the further developments until 1999 by Y.P. Yung (younger brother of Y.W. Yung) and P-H. Sun. Section §4.2 describes the principal highlights of my contribution to the development of EDEN since 1999. Section §4.3 gives an analysis of the operational semantics of EDEN.

4.1 EDEN, chronologically to 1999

4.1.1 The early history

The EDEN tool has received comparatively more attention than the ADM and the DAM machine, and the number of written sources is correspondingly higher. Focussing on the machine rather than applications, however, leads to four primary sources [Yun87, Yun90, Yun93, Yun96] from the two original authors of EDEN which form the basis for much of the analysis in this chapter. This section presents the history of the Eden language and the various implementations. It does not discuss the operation of the definition maintainer, which is discussed in §4.3.

Y.W. 'Edward' Yung was the original author of the Eden language and the first terminal-based implementation which we now call `ttyeden`[1]. The `ttyeden` implementation and hence some parts of the language are based on a tutorial example program included in [KP84], named `hoc`. The explanations that follow are a retrospective rewrite of EDEN history: Y.W. Yung's writings justify and explain EDEN largely independently of `hoc`, but the perspective taken below is useful to highlight the distinction between "standard textbook implementation" and novel definitive features.

Hoc ("high-order calculator") is "an interactive language for floating point arithmetic", written in the C language with the assistance of the `yacc` parser generator

[1] We will use this name throughout to identify the terminal-based implementation of the Eden language, even though the name was not used at this early time.

and the `lex` lexical analyzer. It is similar to the standard UNIX arbitrary-precision calculator utility `bc`. The implementation presented in [KP84] provides a four-function calculator with constants such as `PI`; built-in functions such as `sin`; an assignment statement acting on procedural variables; relational operators and control flow; recursive procedures and functions with arguments, and input/output routines. The following example is combined from several [KP84, p.234, p.245, p.242, p.246, p.274, p.332, p.331] given in the book for use of `hoc` and gives a fairly complete idea of the language facilities available. Keyboard input is shown `like this`.

```
$ hoc
4*3*2
    24
(1+2) * (3+4)
    21
355/113
    3.1415929
x = 355
y = 113
x/y
    3.1415929
x = y = z = 0
sin(PI/2)
    1
proc fib() {
  a = 0
  b = 1
  while (b < $1) {
    print b
    c = b
    b = a+b
    a = c
  }
  print "\n"
}
fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
func fac() {
  if ($1 <= 0) return 1 else return $1 * fac($1-1)
}
fac(7)
```

```
    5040
func ack() {
  if ($1 == 0) return $2+1
  if ($2 == 0) return ack($1-1, 1)
  return ack($1-1, ack($1, $2-1))
}
ack(3, 2)
    29
while (read(x)) {
  print "value is ", x, "\n"
}
42
value is 42
```

Listing 4.1: An interaction with `hoc`

There are significant differences between `hoc` and `ttyeden`. These include:

- The data types supported are very different from `hoc`. The early version of EDEN had no floating point type. Integer, character, string and pointer data types, the special undefined type ('@') and a heterogeneous list type was included to allow the preceding atomic types to be composed into new data types.
- Some further built-in functions were added to `ttyeden`, mainly to manipulate the new data types.
- An interface was developed to the C library, allowing the addition of functions such as `fprintf`. In particular, 41 functions from the 'curses' terminal window handling package were added.
- Various procedural statements were added to extend the available control flow options (`break`; `continue`; `do`; `for`; `switch`; `case`; `default`), implement variables local to a `proc` or `func` (`auto`) and to manipulate the list data type (`shift`; `append`; `insert`; `delete`).

The introduction of the above features enhances the computational power of the resulting tool, but does not change the programming paradigm. From the perspective of this thesis, other extensions of `hoc` entailed in developing `ttyeden` are far

more important. These include mechanisms for managing dependency that have been so seminal in the development of EM that they merit a separate subsection.

4.1.2 Formula variables and actions

The more radical and significant features that were added to `hoc` in developing `ttyeden` are listed below.

- Formula variables;
- Actions (automatically invoked procedures);
- An 'auto-recalculate' mechanism.

The auto-recalculate mechanism is the subject of §4.3 below. At this point, it will suffice to say that the auto-recalculate mechanism implements automatic evaluation of formulae and actions at the correct time. The automatic evaluation can be temporarily delayed by setting the Eden `autocalc` variable to false.

The variables that are present in `hoc` are known in Eden as Read-Write Variables (RWVs[2]). RWVs are assigned to using the `=` operator, using the conventional syntax *identifier* = *expression*. For example:

```
a = a+1;
```

Formula variables (FVs) in Eden are distinct from RWVs. Y.W. Yung has the following to say [Yun90, p.27] about the distinction between FVs and RWVs.

> EDEN supports the concept of formula. A formula definition has the form:
>
> *identifier* `is` *expression* ;
>
> or
>
> F is $\phi(v_1, v_2, \ldots, v_n)$;
>
> The keyword '`is`' defines a formula variable 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 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 redefined). That is, no matter what the values of source variables are, the value of variable F is *always* equal to $\phi(v_1, v_2, \ldots, v_n)$. Thus a formula gives an abstract definition

[2]The term RWV is introduced in [Yun90] — [Yun87] uses the term "value variable".

> 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,
>
> $V = \phi(v_1, v_2, \ldots, v_n)$;
>
> the value of 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 instant of evaluation and stores the result in the variable V.

Actions in Eden build on the procedure- and function-call subroutine mechanism from `hoc`. They are procedures which are automatically invoked when a named FV or RWV changes value. Y.W. Yung [Yun90, p.30] outlines the syntax and motivation for actions as follows.

> 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 '*changed*' 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.

Later, Y.W. Yung [Yun90, p.37] defines the intent of the word *changed*.

> 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.

An alternative way of defining actions which "is sometimes more appropriate" is described in [Yun87, p.13]. The syntax is:

> *variable* `~> [` *procedure_list* `] ;`

The example shown above can be written in this alternate form as:

```
proc display_v { writeln(v); }
v ~> [display_v];
```

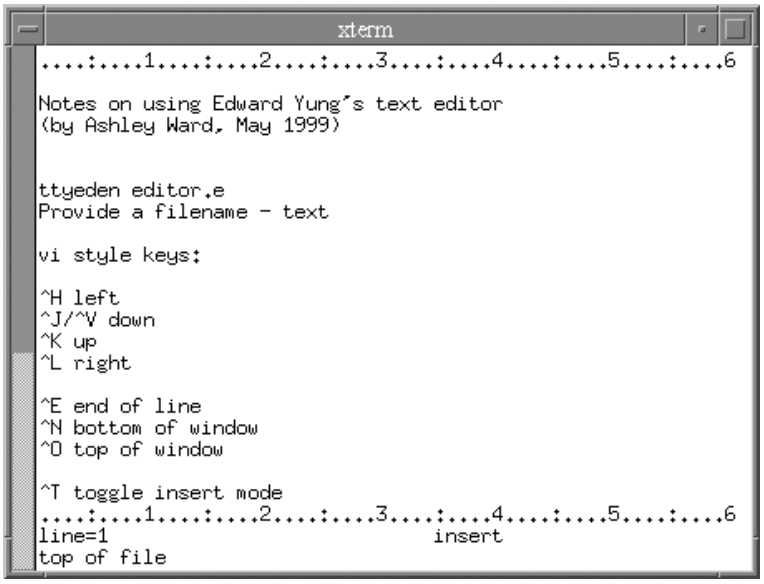


Figure 4.1: *texteditorYung1987* running in `ttyeden-1.52`

Y.W. Yung [Yun87, p.13] suggests that the syntax '`v ~> [display_v]`' can be read as "as `v` is changed, do `display_v`".

4.1.3 *texteditorYung1987*

The first non-trivial definitive model to be written in Eden was the major example given in [Yun87], which I have filed in the empublic archive [WRB] as *texteditorYung1987*. The model implements a terminal-based 'visual' text editor, rather like the standard editor `vi`. The model still runs in the current version of `ttyeden` that I maintain 17 years later, in 2004: `ttyeden-1.52`. Figure 4.1 shows the text editor being used to edit some text that describes how to use it.

In this section, I present an analytical review of the *texteditorYung1987* model in order to demonstrate the principal features of `ttyeden` (the 'purest' Eden implementation) which were briefly outlined in the previous two subsections. It is also interesting to compare this model to the example given in §3.5.4 where the DAM machine is configured to display three character patterns on the screen, the pixels making up the glyphs being dependent on the text data.

Firstly, I show parts[3] of the model in order to demonstrate use of RWVs, FVs,

[3] I only give extracts as the original is some 479 lines long.

```
1  text = [""];
2  line = 1;
3  col = 1;
4  insert_mode = 1;    /* default insert_mode = ON */
5
6  WIN_WIDTH = 60;
7  WIN_HEIGHT = 20;
8  win_left = 1;
9
10 RULER = repeatchar(MAXCOL, '.');
11 for(i = 5 ; i <= MAXCOL; i += 10) RULER[i] = ':';
12 for(i = 10; i <= MAXCOL; i += 10) RULER[i] = char(i/10 % 10 + '0');
```

Listing 4.2: A selection of Read-Write Variables from *texteditorYung1987*

actions, funcs and procs in Eden. The order of investigation represents one way in which an analysis of a definitive model can usefully proceed, unusually starting with the data rather than the actions.

In the final part of this section, I show the powerful potential for the use of dependency, even in this terminal-only variant of EDEN, by introducing some re-definitions to *texteditorYung1987*.

RWVs (Read-Write Variables)

Listing 4.2 shows a selection of Read-Write Variabless that define the text that is being edited, which is initially blank (line 1); the cursor position within the text (lines 2 and 3); and whether the editor is in 'replace' ('over-type') or 'insert' mode (line 4). Next, the size of the terminal window is assigned (lines 6 and 7). The terminal window will sometimes be narrower than the full text, and the editor is able to scroll the window to display just a portion of the text. The RWV `win_left` (line 8) holds the index of the leftmost displayed character.

Figure 4.1 shows a 'ruler' displayed at the top and bottom of the text, which gives an indication of the column indices. The `RULER` RWV string that will be shown as the ruler is initialised by first using a function to create a line of periods (line 10), some of which are then overwritten by the two `for` loops that follow (lines 11 and 12), which introduce the positional markings into the ruler string.

```
1 CUR_LINE is (line > text#) ? "" : text[line];
2 CUR_CHAR is (col > CUR_LINE#) ? ' ' : CUR_LINE[col];
3
4 win_right is win_left + WIN_WIDTH - 1;
5
6 CUR_RULER is substr(RULER, win_left, win_right);
7
8 MAXCOL is 100;
```

Listing 4.3: Some Formula Variables in *texteditorYung1987*

FVs (Formula Variables)

Some examples of Formula Variables are shown in Listing 4.3. FVs define a string containing the current line of text (line 1) and current character (line 2) at the cursor position, with blank substitutions if the cursor has moved beyond the limits of the current text[4]. The index of the rightmost displayed character, `win_right` (line 4), is calculated from the `win_left` and `WIN_WIDTH` RWVs shown in Listing 4.2. The portion of the ruler string which is currently on the screen (`CUR_RULER`, line 6) is calculated by finding the appropriate substring of `RULER`. The last FV here (`MAXCOL`, line 8) is given as an example to show that FVs can simply be defined to literal values, in which case they act like a RWV.

Actions

The text editor model contains many actions, most of which eventually call the 'curses' C library routines to change the terminal window state. Some examples are shown in Listing 4.4. The `~>` syntax is used to cause a change of variable to invoke a procedure call. The listing specifies how a change to the `message` RWV causes (line 18) a call to the `print_message` procedure action (line 1) and then the `update_scrn` action (line 13), which together cause the value of the `message` variable to appear on the bottom line of the terminal. (See the indication "top of file" shown in Figure 4.1 for example.) Any change in the `insert_mode` of the editor, or to the displayed portion of the ruler (`CUR_RULER`), is propagated to the display in a similar way (lines 19,20).

[4]The `#` operator gives the number of items in a list/characters in a string, and `?:` is the ternary 'if-then-else' operator, as provided in the language C.

```
1  proc print_message {
2      if (message == @) return;
3      move(WIN_HEIGHT + 3, 0);
4      addstr(message);
5      addch('\n');
6  }
7
8  proc print_insert_mode {
9      move(WIN_HEIGHT + 2, 33);
10     addstr(insert_mode ? " insert" : "replace" );
11 }
12
13 proc update_scrn {
14     at(line, col);
15     refresh();
16 }
17
18 message  ~>    [print_message,           update_scrn];
19 insert_mode ~> [print_insert_mode,       update_scrn];
20 CUR_RULER ~>   [print_ruler, renew_scrn, update_scrn];
```

Listing 4.4: Some actions in *texteditorYung1987*

Functions

Functions in Eden are very similar to those in `hoc`. The text editor model defines a few utility functions, including the following function that returns the maximum of two provided parameters:

```
func max { return $1 > $2 ? $1 : $2 ; }
```

Procedures

Finally, procedures are used to encapsulate and name state change. Some procedures from the *texteditorYung1987* model are shown in Listing 4.5. The procedure `right` is called to move the cursor one place to the right. If the cursor position exceeds the maximum line length, then the `message` RWV is set to show an appropriate error (line 5) and is then automatically displayed on the terminal by the `print_message` and `update_scrn` actions shown in Listing 4.4.

```
1  proc right {
2      if (col < MAXCOL)
3          col++;
4      else
5          message = "right margin";
6  }
7
8  proc replace_char {
9      if (col <= CUR_LINE#) {
10         text[line][col] = $1;
11         addch($1);
12         right();
13     } else
14         insert_char($1);
15 }
```

Listing 4.5: Some procedures in *texteditorYung1987*

The procedure `replace_char` (line 8) is called when a key is pressed and the editor is in 'replace' (rather than 'insert') mode. The procedure modifies the `text` RWV and calls the curses `addch` routine to make the corresponding change on the terminal. Notice that changes to the `text` RWV are propagated to the terminal in this *ad hoc* manner rather than by using dependency in the manner of the dependent pixels example in §3.5.4. This is probably due to problems with using lists in dependencies, which we shall describe in §5.2.1. The *ad hoc* propagation of change can cause problems in synchronising the EDEN state and the terminal state, as we shall see shortly.

The `edit` procedure is invoked to start the editor. A selection of lines from this moderately long procedure are shown in Listing 4.6. The procedure repeatedly waits to get a key press (the `fgetc` call on line 4) and then calls the appropriate procedure to process it. For example, the `right()` procedure is called (on line 17) when control-L (the key press for 'move cursor right' in the `vi` editor) is pressed.

Whilst the edit procedure is blocked waiting for a key press, the entire `ttyeden` process is blocked, as EDEN is a sequential, single-threaded machine. The edit procedure therefore has exclusive control over EDEN when it is running. The control-D key command terminates the edit procedure (line 14) and `ttyeden` can then be used interactively in the normal way (using the Eden language) again.

```
proc edit {
    auto c;
...
    while (mycbreak() && (c = fgetc(stdin)) != EOF) {
        if (message != "")
            message = "";
        if (c < ' ') {
            switch (c + 'A' - 1) {
...
            case 'D':       /* ^D */
                reset();
                move(WIN_HEIGHT+3, 0);
                refresh();
                return;
...
            case 'L':   /* ^L */
                right();
                break;
...
            case 'T':    /* ^T */
                insert_mode = ! insert_mode;
                break;
...
        } else if (c == 127) { /* DEL */
            delete_char();
        } else if (isprint(c)) {
            c = char(c);
            if (insert_mode)
                insert_char(c);
            else
                replace_char(c);
        }
    }
}
```

Listing 4.6: Highlights from the *texteditorYung1987* `edit` procedure

```
RULERPROMPT is str(text#) // " lines in total";
CUR_RULER is substr(RULER, win_left, win_left+5) // RULERPROMPT //
  substr(RULER, win_left+5+RULERPROMPT#, win_right);

WIN_WIDTH is col + 10;

insert_mode is CUR_CHAR != 'a';

func st {
  auto s, ret;
  ret = [];
  s = symbols("formula") // symbols("var") // symbols("proc") //
        symbols("func");
  while (s != []) {
    if (symboldetail(s[1])[6] != "system")
      append ret, symboldefinition(s[1]);
    shift s;
  }
  return ret;
}
text is st();
```

Listing 4.7: Some possible redefinitions to make to *texteditorYung1987*

Redefinitions

The model does not use a large number of FVs. In this final part of the section, I show how judicious introduction of more FVs to the model can make interesting modifications to the text editor, beyond what Y.W. Yung would have conceived. For example, I introduce FVs where — expecting the data to be exclusively under program control — he used RWVs. As well as demonstrating the potential for change that is not preconceived, the redefinition examples also serve to illustrate various limitations of our current implementation of dependency.

Four examples of the use of dependency are given in Listing 4.7. They perform the following modifications to the text editor:

- Modify the ruler to indicate the total number of lines in the text;
- Make the terminal window width dependent upon the cursor position;
- Make the editor insert/replace mode dependent upon the current cursor character;
- Make the editor show its own Eden code.

The first pair of definitions (lines 1–3 of Listing 4.7) change the ruler to include information about the total number of lines in the edited text. The `//` operator is used to concatenate strings in Eden. Two definitions are used to cope with the fact that the number of lines in decimal is a variable number of characters long, yet `CUR_RULER` needs to be exactly `win_right - win_left` characters long. After these definitions are introduced, the ruler is automatically updated with the new information when the text is modified. Note that the `RULERPROMPT` variable (and hence the `CUR_RULER` variable) need only be updated when the number of items in the Eden `text` list variable changes, but it is actually updated whenever the `text` variable changes in any way.

The ruler was defined as a FV in the original model. RWVs can also be changed to FVs with unusual but perhaps useful results. For example, the length of the lines in the display can be defined to be always ten more than the current horizontal cursor position (line 5). Moving the cursor to the right then causes the display to 'expand'. Above, we mentioned the *ad hoc* way in which changes are propagated to the terminal by the model: the `replace_char` procedure shown in Listing 4.5 calls the curses `addch` routine as a side-effect. The internal state of the `text` variable is not linked to the external state on the terminal by dependency. As a result, moving the cursor to the left does not immediately contract the terminal window, as there is no action in the model to output clearing blanks to the right of the current display.

The editor can be made to automatically change to 'replace' mode when the cursor is over an '`a`' character and insert mode otherwise, with the redefinition shown on line 7 of Listing 4.7. This example is given partially to illustrate a problem, as follows. If `insert_mode` is defined as a FV, as shown on line 7, and then the user uses the "toggle insert mode" key press (control-T), the `edit` procedure will make a procedural assignment (using '`=`') to the `insert_mode` variable (on line 21 of Listing 4.6), changing it back from a FV to a RWV. This illustrates that dependencies can be easily destroyed with a careless procedural assignment in EDEN.

Finally, the function and definition shown on lines 9–21 of Listing 4.7 cause the text editor to show its own Eden code[5]. Further developments involving "in itself" aspects of Eden are described in §4.2. This particular example is given here in order

[5]The `st` function shown requires a facility for identifying 'system'-defined variables which is available in version 1.52.

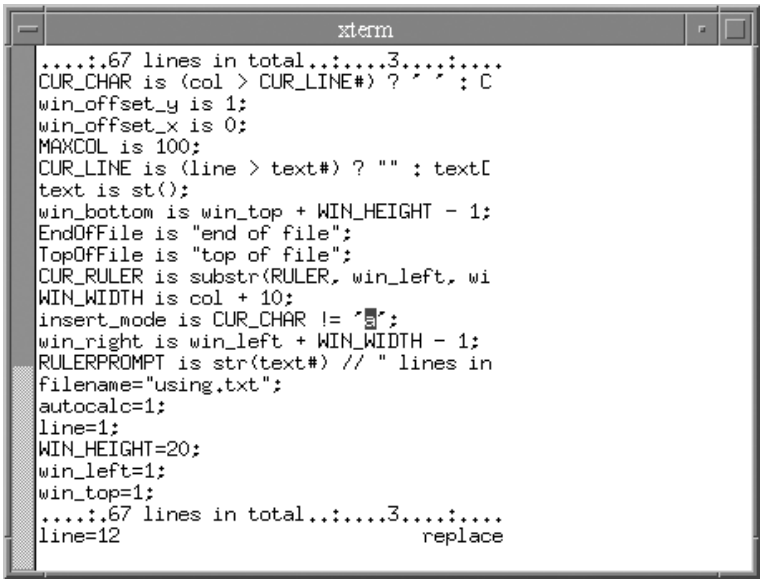


Figure 4.2: *texteditorYung1987* with some example redefinitions

to illustrate two issues. Firstly, note that although this example successfully displays the model state in the text editor, the text cannot be edited, as the dependency `text is st()` implies only a one-way constraint. Secondly, note that if the text editor model state changes (when a redefinition is made for example), the text editor does not automatically re-evaluate the `st()` function and display the new state. There is no way to specify the necessary dependency in the current Eden.

The modified text editor with these four sets of redefinitions is shown in Figure 4.2.

4.1.4 DoNaLD, SCOUT and ARCA pipeline translators

Although Eden is a powerful general-purpose definitive language, we desire domain-specific definitive notations to assist with our modelling. 'Pure' definitive notations comprise only (re)definitions, of the form *id* `=` *expr*. They are based purely on dependency. Agency is limited to that of the modeller, so a pure definitive notation provides a 1-agent modelling environment.

The first definitive notation to be implemented with EDEN was DoNaLD, a Definitive Notation for Line Drawing. DoNaLD was specified before the advent of EDEN, in [BABH86]. Y.W. Yung was the author of the original implementation,

and describes it in [Yun90, §6]. Y.P. Yung designed and implemented the SCOUT definitive notation (which describes SCreen layOUT) using the same strategy for implementation. The strategy for implementing a definitive notation in EDEN, codified by Y.P. Yung in [Yun93, §6.3.1], is quoted below as it provides the explanation behind the resulting translator. It is also a nice example of incremental construction and "human computing", where a process is first attempted manually and later automated.

1. Derive a scheme for translating [definitive notation] variable names into Eden variable names. For example:

DoNaLD name	Eden name
`table`	`_table`
`table/drawer`	`_table_drawer`
`table/drawer/width`	`_table_drawer_width`

2. Emulate the data types and operators using Eden data types and user-defined functions. Almost inevitably this will make use of the list structure in Eden because list is the only complex data type in Eden. For example:

DoNaLD type	Eden type
`integer`	`integer`
`point`	`['C', integer, integer]`
`line`	`['L', point, point]`

DoNaLD operator	Eden operator/function
`div`	`/`

`+ (vector sum)`	`func vector_add {`
	`  para p1, p2;`
	`  return ['C', p1[2]+p2[2],`
	`  p1[3]+p2[3]];`
	`}`

3. The underlying algebra of the target notation has been implemented through steps 1 and 2. To complete the implementation, the required implicit actions are emulated using Eden's user-defined actions. For example:

DoNaLD code	Eden action specification
`integer i`	No action
`point p`	`proc P_p: _p { plot_point(&_p); }`
`line L`	`proc P_L: _L { plot_line(&_L); }`

   Notes: No action is required for integer `i` because integer variables do not have any graphical representation in DoNaLD. The `&` operator is similar to that in the C language: it returns the address of the variable. `plot_point` and `plot_line` are Eden (user-defined) procedures which do the plotting.

4. Write a preprocessor to translate scripts in the definitive notation into Eden in the way implicitly defined by steps 1 to 3.

The original DoNaLD implementation communicated with EDEN through a UNIX pipeline. Y.W. Yung [Yun90, p.89] gives the details.

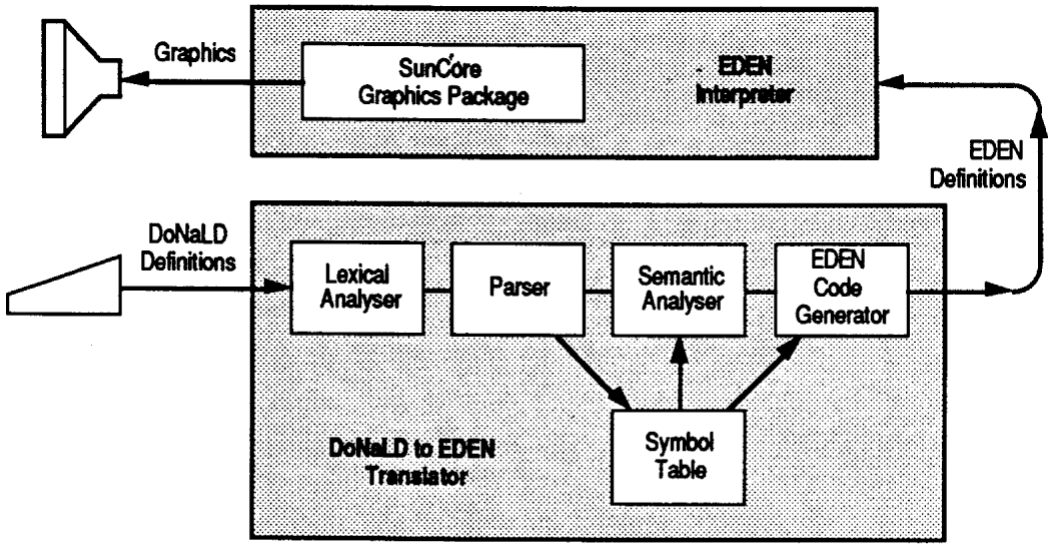


Figure 4.3: The original DoNaLD system (from [Yun90, Figure 6-1])

The following UNIX shell script, Donald, forms the complete DoNaLD system:

```
donald.translator | eden -n init.e
```

where `donald.translator` and `eden` are the DoNaLD to Eden translator and EDEN interpreter respectively.

The `donald.translator` is a filter where DoNaLD specification is read from the standard input file and the EDEN code is output to the standard output file (i.e. pass to `eden`).

The `-n` option tells `eden` to run in non-interactive mode (i.e. no prompt). The file `init.e` stores all supporting EDEN codes which is loaded before `eden` reads the standard input (i.e. the output of `donald.translator`).

The `donald.translator` program was constructed using the conventional `lex` and `yacc` tools. The actual graphics were generated by EDEN procedures which called the SunCore C library package available at the time. The whole system is illustrated in Figure 4.3.

Much work on definitive notation translators was performed at this time: the DoNaLD translator was later enhanced by Chan [Cha89] and Parsons [Par91]. The original SCOUT and ADM (see §2.2.2) translators were implemented by Y.P. Yung in a similar manner to DoNaLD. Stuart Bird wrote the first ARCA[6] to Eden

[6] A notation for describing Cayley diagrams.

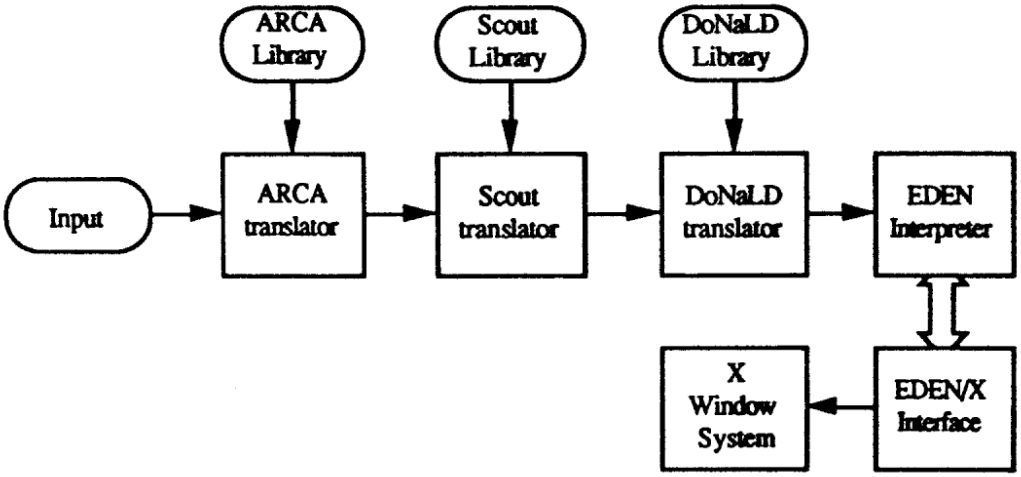


Figure 4.4: ARCA, SCOUT and DoNaLD combined on a pipeline (from [Yun93, Figure 6-5])

translator *arcaBird1991*, again implemented in a similar manner, which was later modified by Y.P. Yung and then myself (*arcaWard2002*).

By 1992, it was possible to integrate all these definitive notations using UNIX pipelines, and an 'EDEN/X' (EX) interface had been constructed to display results using the new X Window client-server graphics system. Y.P. Yung named the result "the SCOUT system". The pipeline is illustrated in Figure 4.4 and described in the quote below from [Yun93, p.95].

> The SCOUT system is implemented in a UNIX environment. The user's input is pipelined through a series of ARCA, SCOUT and DoNaLD filters (the ordering of the filters is not important) which translate ARCA, SCOUT and DoNaLD definitions into Eden definitions. These definitions together with the functions and actions in the libraries are interpreted by the EDEN interpreter. Graphical outputs are generated by an EDEN/X interface which is actually an X Window client. When the graphics display needs to be updated (i.e. some EDEN actions generate graphics output), the EDEN interpreter will interact with the EDEN/X interface, which in turn will interact with the X Window server to produce graphics images.

Note that there is only one input, and so a mechanism must be devised to control which translators in the pipeline will act on the input. Y.P. Yung [Yun93, p.99] describes the details in the quote below.

> In order to incorporate several definitive notations into a single system, the SCOUT system uses a method similar to the way the preprocessors of `roff` (the standard UNIX text formatting language) works. A block of definitions of a definitive notation is preceded by a declaration of the notation name. For example:
>
> ```
> %scout
> ...
> SCOUT definitions
> ...
> %donald
> ...
> DoNaLD definitions
> ...
> ```
>
> The individual translator will translate only the lines from the notation declaration downwards until the declaration of another definitive notation is reached; the remaining lines are untouched. In this method, all the translators are running independently of one another.

Many different ways of organising the communication between individual translators of definitive notations are represented in the versions of EDEN developed by Y.W. Yung and Y.P. Yung between 1989 and 1996. Several key issues[7] are raised. These relate to the extent to which EDEN:

- can itself generate input for translators;
- can hold an image of the overall system state for use in the interface to the modeller;
- supports interleaving of actions on the part of automated and human agents, and
- permits the experimental development of new notations.

The next section discusses the role played by some of these issues in Y.P. Yung's development of `tkeden`. Section §4.2.4 describes a generalised notations framework that I have developed that has the potential to address all the above issues.

[7] At this time, efficiency was also a significant concern, but this is no longer relevant largely, due to "Moore's law" (see §4.1.7) and the fact that typical Eden models are small as the bulk of the definitions in a script are usually hand-crafted.

4.1.5 The early tkeden

In developing `tkeden`, Y.P. 'Simon' Yung set out to integrate several definitive notations into one tool, using the then-new Tcl/Tk graphics system [Ous94]. He explains the motivation and results in [Yun96, p.7].

> The shortcoming of [the SCOUT] system is that interactions between different modules are restricted. The pipeline cannot be wrapped round. Therefore, EDEN cannot easily and efficiently generate DoNaLD or SCOUT definitions and pass them to the appropriate translator located at the front of the pipeline. Also, the separation of the translators, the EDEN interpreter and the graphical interface means that information about the overall system state cannot easily be examined.
>
> In order to solve these problems, the integrated environment `xeden` and its successor `tkeden` are developed. Xeden combines the DoNaLD translator, the SCOUT translator, the EDEN interpreter as well as EX into one unit. Apart from the speed improvement, `xeden` can then use the existing EDEN `execute()` command to evaluate strings representing a DoNaLD or SCOUT script. The DoNaLD `graph` function[8] is later implemented based on this newly established capability.
>
> `tkeden` is developed from `xeden`. Instead of using the core of our home-grown program EX as the interface to X, we make use of Tk as the graphics and event driver. The advantages of using Tk include:
>
> 1. faster graphics (compared to EX),
> 2. 3D look graphical interface: SCOUT windows are now 3D,
> 3. easier to build multi-window graphical user interface,
> 4. easier to extend our graphics notations because Tcl/Tk is a script language,
> 5. better connectivity to other programs because Tk has a builtin interprocess communication mechanism,
> 6. possibility of porting to non-X environment due to the portability of Tcl/Tk.
>
> ... `tkeden`, over and above `xeden`, provides the following features:
>
> - different views of the definition stores;
> - save and load the current set of definitions;
> - view and save the history of interaction.

The tool that Y.P. Yung produced, `tkeden`, included the DoNaLD and SCOUT notations. Figure 4.5 shows an early version of `tkeden` running *cruisecontrol-Bridge1991*, which uses Eden, DoNaLD and SCOUT to model a vehicle with a cruise control facility progressing up and down a sinusoidal hill.

[8] A higher-order definition feature.

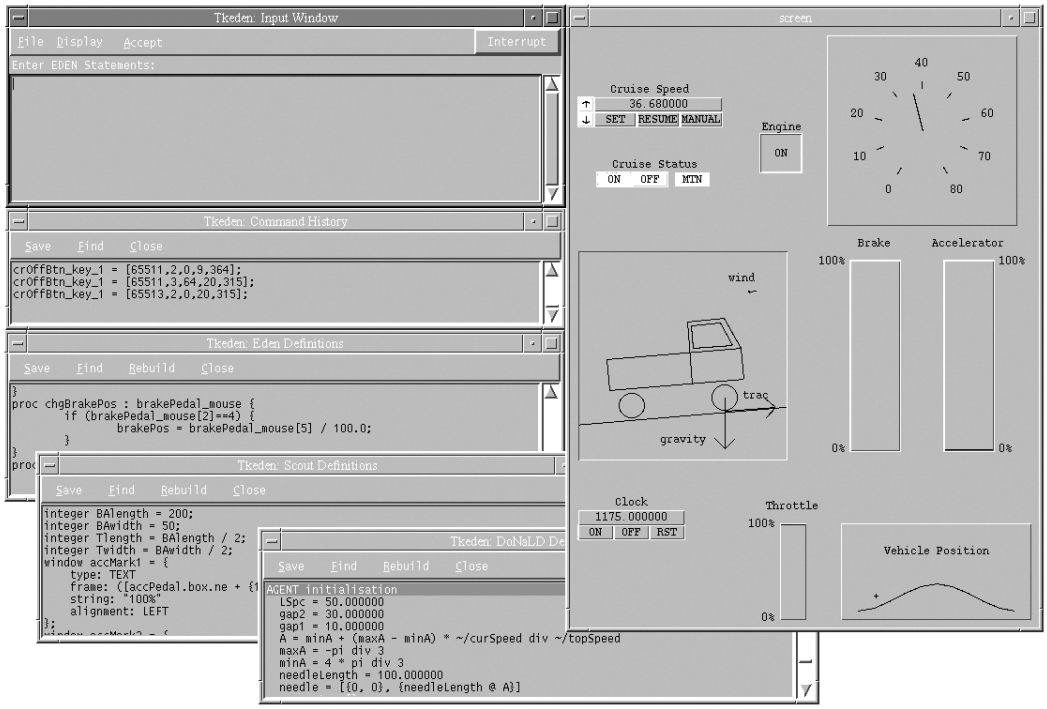


Figure 4.5: `tkeden-dec151997` running *cruisecontrolBridge1991*

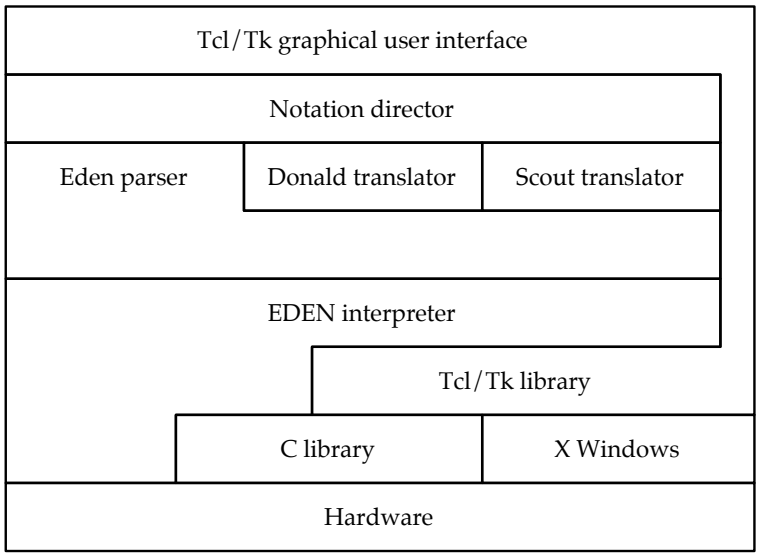


Figure 4.6: `tkeden` architecture

The approximate architecture of the original tkeden, viewed from a high level, is illustrated in Figure 4.6 (by this author). The graphical interface that can be seen in Figure 4.5, comprising the Input Window, the SCOUT screen and other windows, is implemented using Tcl/Tk. Input text is directed towards the appropriate translator which is selected by %notation directives in the input, as before. The DoNaLD and SCOUT translators are essentially the same as before: they translate input into Eden output, but the input/output mechanism has been changed. In the previous architecture, the translators were concurrent processes connected to the EDEN interpreter via a UNIX pipeline, but in this architecture, the appropriate translator routine is sequentially invoked with the input text and is expected to call routines to add Eden output to a queue for processing. Lower down in the figure, the EDEN interpreter calls the Tcl/Tk library routines in order to change the state of the screen. Tcl/Tk in turn calls the X Windows libraries to effect visual change.

4.1.6 Distributed tkeden: dtkeden

P-H. 'Patrick' Sun was the third person to undertake major work on EDEN, producing a variant named dtkeden, capable of distributed client-server communication of redefinitions. Sun describes the results in his PhD thesis [Sun99]. The distributed features in dtkeden are not examined in detail in this thesis as the analysis presented later on proceeds bottom up, starting from the sequential von Neumann machine hardware and the procedural C language in which the current EDEN tool is largely written. Chapter 5 of this thesis presents a framework for dependency maintenance which considers concurrency — future research will be required to reconcile this framework and dtkeden. It is probable that such a reconciliation will require a fundamental revision of the mechanisms for inter-process communication in dtkeden, which are currently implemented without formal attention to synchronisation.

The first dtkeden case study was *claytontunnelSun1999* — a model of the railway accident that occurred in the Clayton Tunnel on the Brighton to London Victoria line in 1861. The accident is described in Rolt's "Red for Danger" [Rol82] and the case study is described in [Sun99] and [Bey99]. This model is again constructed using the Eden, SCOUT and DoNaLD notations. The dtkeden server shows "God's eye view", where all the observables are objectively presented — see Figure 4.7.

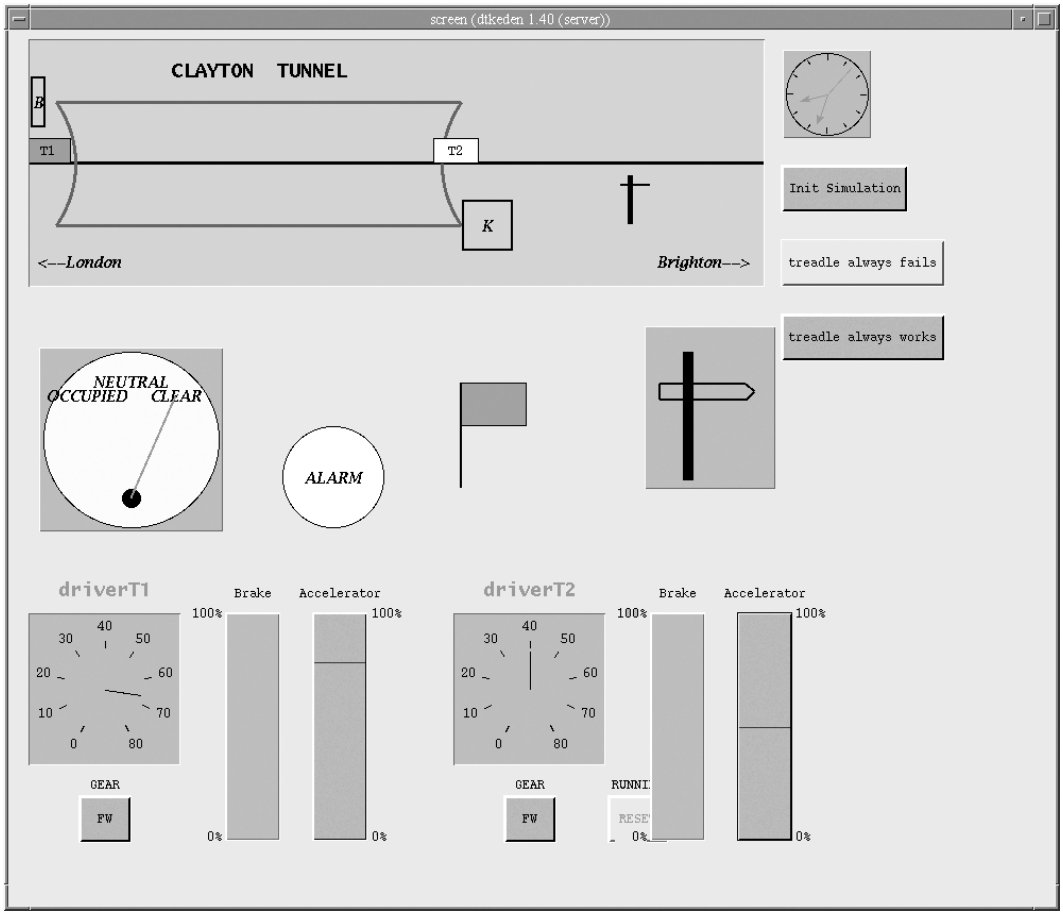


Figure 4.7: "God's eye view" in *claytontunnelSun1999* on the `dtkeden` server

Figure 4.8: `dtkeden` and *claytontunnelSun1999* in use with school children during the ACE week in January 1999

The *claytontunnelSun1999* model requires five `dtkeden` client computers and human users, who play the roles of three train drivers and two signal men respectively, each seeing only their subjective perception of the situation. As part of our investigation into EM for educational applications, we introduced the model to several groups of school children in January 1999 (see Figure 4.8). As discussed by Roe [Roe03], the educational agenda has since motivated much model construction and the creation of the empublic archive [WRB].

4.1.7 Fifty versions: an overview

Figure 4.9 shows the growth of the EDEN code over the 17 year project period so far. The measure shown is a simple count of non-blank lines of code. An attempt has been made to avoid counting automatically generated sources (e.g. output from `yacc`). The count has been divided into that originating from C language files, Tcl files and Eden files[9]. All the source code that remains available has been measured, but only certain selected data points are shown.

[9]File types were determined as follows: files with `.c`, `.h`, `.y` and `.l` extensions (but not generated files) were counted as C language files; the `.tcl` extension counted as a Tcl file, and `.e` and the more recent `.eden` extension counted as Eden files.

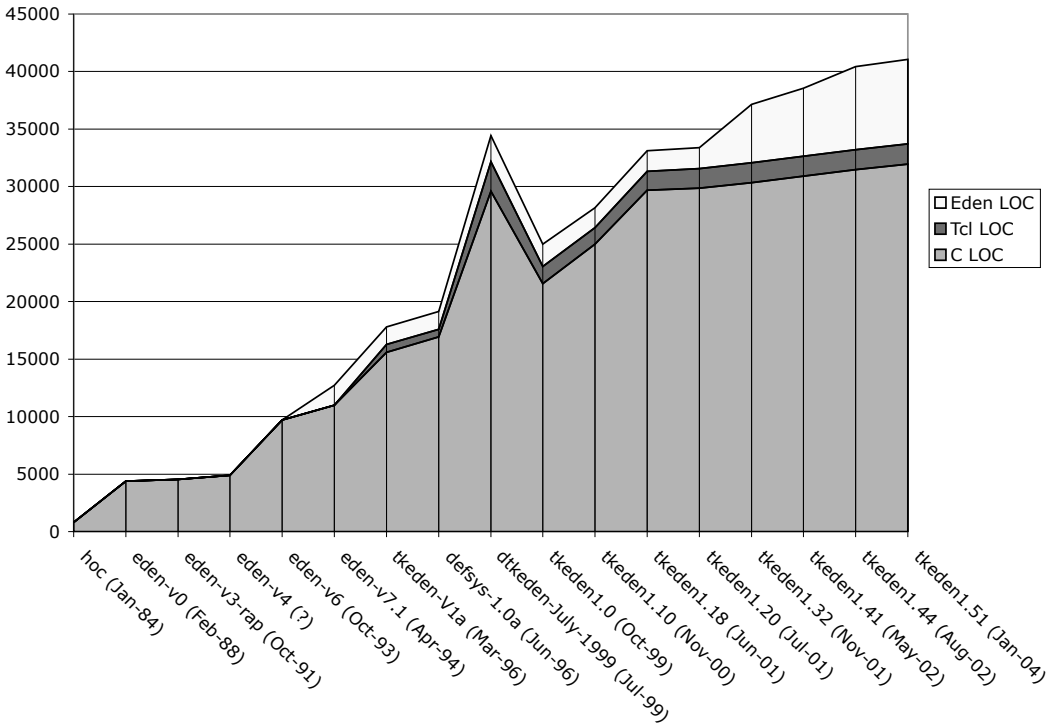


Figure 4.9: Code size of EDEN from 1988 to 2004 (1984 `hoc` shown for comparison)

The chart starts with the `hoc` source code for comparison and moves on to the earliest remaining EDEN source, `eden-v0` (which is actually not quite the earliest version as printed in [Yun87]). The EDEN sources from the version v0 through v7.1 to tkeden-V1a were developed by Y.P. Yung between 1988 and 1996. According to his documentation, it appears that the growth shown between v4 and v6 is due to merging of the EDEN/X interface into the main code base, creating `xeden`. The EDEN `todo()` command (see §4.3) was apparently added in v6.1 and the SCOUT and DoNaLD translators were apparently integrated into `xeden` in v7: this can be seen in the chart as the introduction of 1725 lines of EDEN code in v7.1. The earliest remaining `tkeden` source code appears to date from March 1996 and includes 666 lines of Tcl code. The `dtkeden` data point is atypical when shown amongst the general progression. This version of the source has much duplication between variants of files for server and client. The duplication was reduced when I merged the `dtkeden` source code with the original `tkeden` source, making `ttyeden`, `tkeden` and `dtkeden` compile-time options from a single source. The remainder of the chart

illustrates changes and extensions since 1999, some of which are outlined below.

Reflecting on this period of computing history: the personal computing world was in its infancy in 1987. For example, the first Apple Macintosh was sold only three years before in 1984. Hennessy and Patterson [HP03, p.3] plot the growth in microprocessor performance from 1984-2000 and find that it has increased annually by 50%. The performance available now (in 2004) is therefore nearly 1000 times greater than was available in 1987.

An illustration of the effect of this change in performance on EDEN is given by Y.P. Yung and Farkas's model of a billiards game *billiardsYung1996*, which includes an element of simulation. The simulation uses an integration step-size that is adaptively dependent upon the ball speed [FBY93] in order to precisely determine the point of collision between balls. On the machines common in 1996, the simulation ran rather slower than real-time. The dynamic adaptation of the step-size also caused the ball velocity (as observed in the graphical model) to slow dramatically as another ball was approached. Y.P. Yung therefore added a facility to record the history of redefinitions as the simulation of a shot occurred, and later 'replay' it back, on demand. The replay of redefinitions is not computationally intensive and so can be linked to real-time, causing the replay to occur at a continuously realistic speed. Now in 2004, however, when run on a recent machine, we find that the initial billiards simulation occurs *faster* than real-time — the replay facility slows the simulation down to a continuously realistic speed. The continued progression of CPU performance, which is still currently following "Moore's law" [Moo65], leads to improvements in our models and tools on the basis of mere maintenance on our part. The continued progression is partial vindication of our prioritisation of meaningful state over raw performance. It seems likely that the performance increases will continue in the short term (to approximately 2009), but research breakthroughs are required in most technical areas if progress is not to stall in the long term [Wil02, AEWJ+02, itr].

This increase in performance has been accompanied by a growing desire to run EDEN on multiple platforms. Richard Cartwright first compiled the software for Linux. Ben Carter completed a port of `tkeden` to the Windows platform in early 2000. I have been maintaining these ports and also completed an initial port to

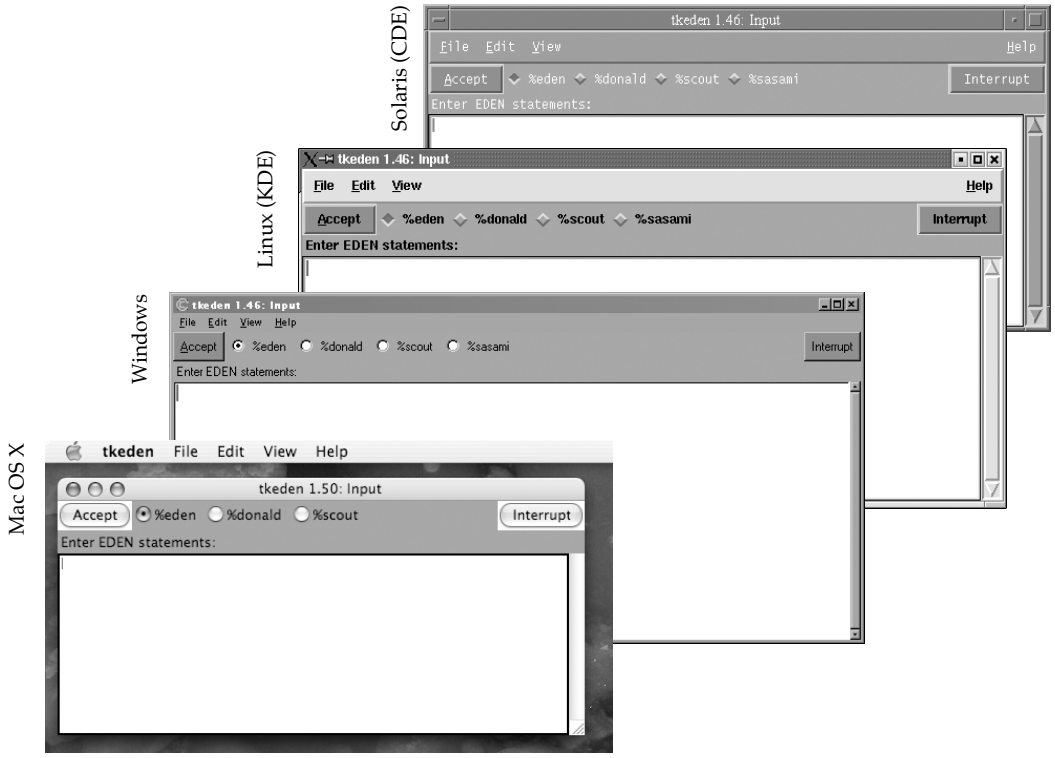


Figure 4.10: `tkeden` running on Solaris, Linux, Windows and Mac OS X

Mac OS X in early 2002, releasing a slightly more mature version in March 2003. Figure 4.10 shows the different appearances of the interface on the different platforms. In particular, the arrangement of the menus vary on each platform in order to conform to the differing interface guidelines on each.

Maintaining the tool on the different platforms makes developing the software more difficult, but it is a necessary part of attracting and retaining an audience for the tool.

The audience for the software has grown since the creation of EDEN. Presently, the vast majority of the EDEN user population is made up of students and staff within the University of Warwick Department of Computer Science. I set up and currently maintain an installation of the EDEN software for those local users. All previous versions are kept in the installation to aid regression testing. In November 2001, I added logging of software runs from the local installation. The time, date, user identity, host name, process identifier, tool variant and version, elapsed (wall

Month	Runs of EDEN	Unique users	Total CPU time (in hours)	Total CPU time / users (in mins)	Downloads via FTP				
					Win	Linux	Solaris	Source code	Docs
200111	141	42	17.2	24.6	23	1	0	0	2
200112	781	124	4.4	2.1	94	14	0	17	157
200201	124	9	64.6	430.5	3	0	0	0	0
200202	380	9	65.8	439.0	9	0	0	1	4
200203	229	9	20.5	136.4	9	0	0	0	5
200204	277	16	7.9	29.6	3	0	0	0	0
200205	355	11	4.9	26.6	21	3	2	6	9
200206	217	33	1.9	3.5	32	2	0	7	10
200207	85	4	0.5	7.1	7	0	0	3	5
200208	133	6	9.4	93.7	6	2	4	3	0
200209	69	5	0.6	7.4	5	1	0	0	7
200210	869	40	23.4	35.1	51	17	12	13	25
200211	3267	167	18.2	6.5	230	46	6	52	243
200212	248	19	5.9	18.6	12	8	3	8	49
200301	1826	16	3.5	13.1	19	0	0	1	4
200302	4044	19	25.0	79.1	3	2	0	2	0
200303	3588	20	61.0	182.9	4	2	0	1	2
200304	1892	13	9.4	43.6	531	98	27	114	522
200305	273	28	19.0	40.8					
200306	161	27	0.3	0.7					
200307	276	8	4.6	34.8					
200308	182	3	1.3	25.6					
200309	297	7	33.7	289.1					
200310	806	47	240.0	306.4					
200311	1280	109	174.6	96.1					
200312	503	18	5.9	19.7					
200401	1454	33	17.6	32.1					
	23757		841.2						

Figure 4.11: Data from logging of local `tkeden` usage and downloads

clock) time, user and system CPU time and the final exit status are all collected. As this logging is potentially intrusive, users are automatically asked to confirm their acceptance of this logging when they first run EDEN. The data collected is presented in Figure 4.11, summarised into months. The data shows that there is a continual background level usage of the tool, corresponding to use by staff and research students. The usage peaks when the software is used in undergraduate teaching: it has been used by many students as a basis for a third year project, on a large database systems level 2 module (producing the large peaks that can be seen in the numbers of unique users in November/December each year) and in an "Introduction to EM" level 4 module.

The usage data shown is usage on the local departmental systems only and does not reflect the additional usage of the software on users' personal machines, which is increasingly popular. This is shown in Figure 4.11, which along with the usage

information, also includes information from logging of FTP downloads, summarised into counts of types of download per month[10].

4.2 Developments to EDEN since 1999

Since taking responsibility for the maintenance of `tkeden` in 1999, I have overseen some fifty released incremental versions of the software: approximately one release per month on average. This is in keeping with the open source "release early, release often" philosophy described by Raymond [Ray], which attempts to create an active feedback loop, assisting in testing the software and driving requirements. There have been many strands to the development, many taking significant effort, but it is not possible to give a full description here[11]. The following material presents the most coherent strands. Subsections §4.2.1 to §4.2.5 inclusive relate to my aim of (re)constructing the tool "in itself". Subsection §4.2.6 shows an environment I constructed in order to use the EDEN tool as a visual aid to support complete presentations, including multiple demonstrations. This environment incidentally made it obvious that multiple models can be 'run' simultaneously in one instance of EDEN. Subsection §4.2.7 briefly describes improvements I have made to the EDEN interface. The subsection includes a description of a definitive partial reconstruction of the `tkeden` interface by Roe, within the presentation environment. The reconstruction is another interesting "in itself" prototype. The final subsection §4.2.8 describes several experiments we have performed that interface EDEN to other systems. The experiments show that dependency can be used successfully in this manner, but issues of concurrent synchronisation are raised — in part due to the single-threaded sequential nature of the EDEN machine, which is explored in the final section §4.3.

4.2.1 The beginnings: Sasami, motivating "in itself"

Figure 4.9 shows that in 2001, a decision was made to prefer to extend the tool by adding Eden code over C code. As of version 1.51, the Eden code constitutes 18%

[10]The ability to use HTTP downloads was added around March 2003, providing a choice of download facility but making the FTP statistics invalid for comparison, and so they are not shown from this point onwards.

[11]For such details, consult the 'change log' file at [Wara].

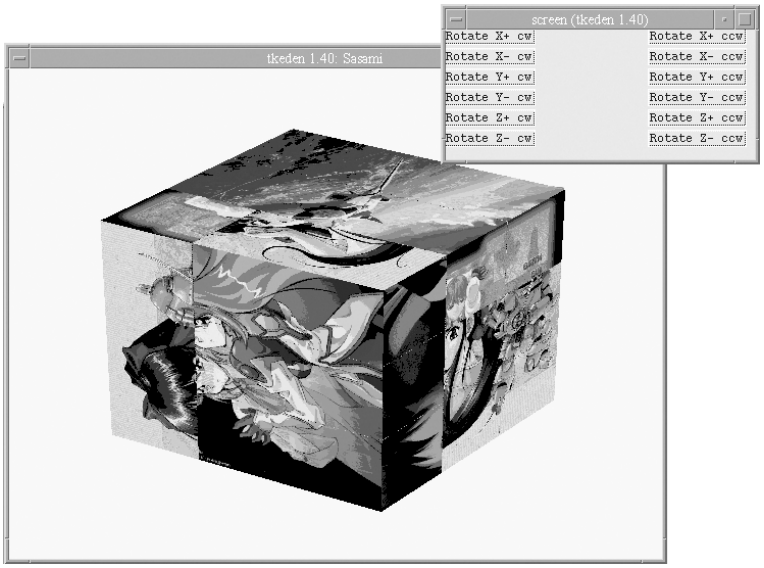


Figure 4.12: The Sasami Rubik's cube, *rubiksCarter1999*

of the total. The motivation for this decision originates with Sasami.

Sasami was the fourth notation (additional to Eden, DoNaLD and SCOUT) to be integrated in the `tkeden` tool. It is a notation for 3D graphics and is implemented using the currently standard OpenGL 3D graphics library. Sasami is the final year project work of Ben Carter [Car00], who was also responsible for the initial `tkeden` port to Windows. The first sizeable case study was *rubiksCarter1999*: a 3D model of Rubik's cube. Figure 4.12 shows the cube after some rotations have been applied using the buttons in the SCOUT interface window at the top right.

The Sasami implementation is technically similar to the DoNaLD and SCOUT implementations. A translator converts statements written in the Sasami notation to Eden code. The Sasami notation is a simple one which is similar to the DAMscript code described in §3.3.2. Parsing it requires no semantic analysis, and so the translator is implemented directly in C rather than with the `yacc` parser generator. For example, the basic primitive in Sasami (as in OpenGL) is the coplanar convex polygon, which is described by a list of vertices (points in Cartesian 3D space). To create a Sasami vertex which can later be used in a polygon vertices list, the `vertex` definition is used. Such a definition has no explicit `is` primitive; it takes the form of the keyword `vertex` and a symbolic name, followed by x, y and z

```
v=1;
_sasami_vertex_1_x is (a+10)/20;
_sasami_vertex_1_y is -c;
_sasami_vertex_1_z is 1.0;
proc _sasami_vertex_mon_1 :
    _sasami_vertex_1_x,
    _sasami_vertex_1_y,
    _sasami_vertex_1_z {
  sasami_vertex(1,_sasami_vertex_1_x,
                  _sasami_vertex_1_y,
                  _sasami_vertex_1_z);
};
```

Listing 4.8: EDEN output corresponding to the Sasami definition `vertex v (a+10)/20 -c 1.0`

coordinates. Each coordinate is described by an Eden expression. For example, the following is a valid Sasami definition:

```
vertex v (a+10)/20 -c 1.0
```

Sasami notation is translated to Eden code. At low levels, Sasami uses unique integer identifiers (which are incremented each time a Sasami object is created) rather than symbolic names. The Eden code therefore contains some indirection from symbolic name to a numeric identifier. Otherwise, the translation is similar to that done by the DoNaLD translator: definitions are created to represent dependencies between objects described at the Sasami level. An Eden action is then created for each Sasami object in order to mediate changes of definitive state to the graphical display. The Eden action calls built-in functions added to EDEN especially for Sasami. For example, the Sasami `vertex` command above is translated into the Eden code shown in Listing 4.8.

The Eden action here calls the new `sasami_vertex` EDEN built-in. This is another part of Sasami, corresponding to the Tcl/Tk graphics subsystem used by DoNaLD. This part of Sasami contains a store of objects referenced by numeric identifiers. Sasami routines call OpenGL routines when rendering is required and read from this store. The data flow is illustrated in Figure 4.13[12].

[12]When porting Sasami to UNIX in version 1.13, I introduced a delay in the bottom-most edge shown in the figure. Now, the graphics hardware is only invoked at 'Tcl_Update' (see Figure 4.38), effectively grouping multiple changes of Sasami state into blocks.

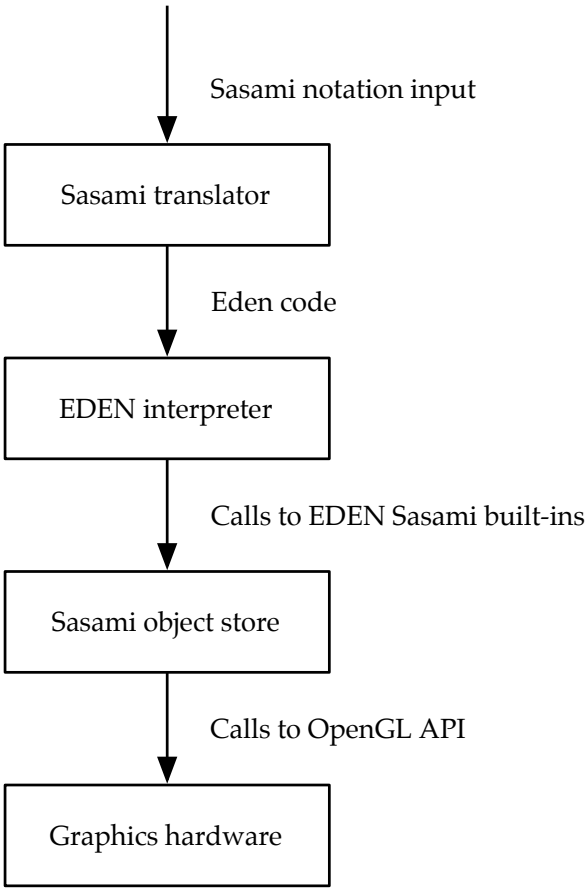


Figure 4.13: Data flow in Sasami

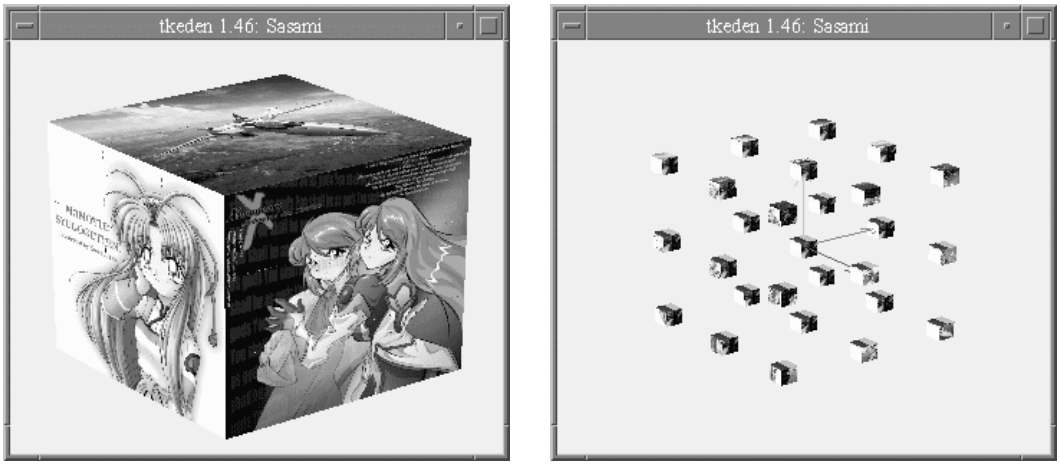


Figure 4.14: The effect of changing the `small_cube_size` variable in *rubiks-Carter1999*

The rotations performed on the Sasami Rubik's cube operate by reassigning cube coordinates and so do not demonstrate the use of dependency[13]. However, the model does contain the Eden variable `small_cube_size`, which controls the size of the distinct cubes through the use of dependency. Figure 4.14 shows the effect of redefining the value from 0.5 to 0.1.

OpenGL is a cross-platform standard but it does not provide an abstraction of the window system. Initially, this part of Sasami was coupled to the Windows platform. In version 1.13, I reengineered this part using the open source 'Togl' package, which provides an OpenGL Tcl/Tk widget. This allowed Sasami to operate on UNIX as well as Windows[14] and should allow integration of the Sasami window into the SCOUT environment in future.

Sasami has now been used in several models (*sasamiexamplesCarter1999*, *rubiksCarter1999*, *billiardsCarter1999*, *room3dsasamiCarter1999*, *cubesymWong2001*, *platonicsolidsBirch2001*, *3doxoRoe2001*, *pyramixRoe2001*, *carparkingsimMcHale2003*). Although these models have been successfully constructed, each has encountered problems with instantiating multiple instances of similar state. This has been solved in each case by writing a program that generates model code. Sometimes the generator program is written in C. In other instances, the generator program is written in Eden and calls the `execute()` command to instantiate state. Either solution presents problems when it is desired to make redefinitions to the model later. This problem seems especially acute when using Sasami. The notation may well be too low-level: it does not have the algebraic properties of DoNaLD or the (unimplemented) definitive notation CADNO (Computer-Aided Design Notation — for which a preliminary design is proposed in [Bey89b]).

Adding Sasami in version 1.7 increased the number of lines of C code, as Figure 4.9 (on p.219) shows. Adding functionality to the EDEN tool by extending the procedural code in this way causes several problems. The following problems are symptomatic of compiled procedural languages generally:

- The tool needs to be recompiled when changes are made. Although this process is faster than in 1987, it is still slow.

[13] It may be possible to model rotations of the individual cubes using dependency by modelling the internal armature mechanism (the 'core' — see [Cao02]) of the cube.

[14] Work on porting Togl to Mac OS X is underway.

- Once the code is compiled, it is then fixed. The audience cannot then modify the tool if it does not suit their needs.

- It is difficult to fully test the result. Ideally all possible behaviours would be exercised (black-box testing). This is clearly an unbounded problem in a general modelling tool. A smaller aim would be to exercise all the lines of code (white-box testing). This requires reverse engineering the control flow into a set of test cases which provides complete code coverage. The tests must be revised when the control flow is modified.

- Modifying the procedural code has the possible side effect of changing the behaviour of the tool. This might then cause some previous models to behave differently, and we have broken backwards compatibility.

Additionally, although C has many advantages when implementing a low-level virtual machine (see §4.3.5), it has disadvantages when used for constructing a translator string processor:

- String handling in C requires explicit management of memory allocation and deallocation.

- C does not (by default) perform range checks on buffer accesses.

Both of these attributes of C have led to problems in EDEN that were difficult to locate and resolve. Checks for problems with memory management can be made automatically by linking with a debugging memory allocator library (for example, 'dmalloc' [Wat]), and I have used this technique to assist in debugging EDEN. However these techniques cannot be used routinely as the debugging allocator greatly slows down execution.

Further, modular construction leads to multiple copies of identical state within different modules. The three copies of Sasami state shown in Figure 4.13 — one copy in the EDEN symbol table, one in the Sasami object store and one in the OpenGL machine are one example. In the Sasami case, this is a waste of memory, which may or may not present a problem. The current DoNaLD system presents a similar example with what might be considered worse consequences. The DoNaLD notation

is strongly typed. However, the strongly typed facilities of DoNaLD are somewhat contrary to our needs today when we consider the "incremental construction and revision" motivation for our tools. As a consequence of strong typing, the DoNaLD translator maintains its own symbol table (as shown in Figure 4.3), in order to perform type checks on DoNaLD expressions and resolve overloaded operators. But the DoNaLD symbol table is inaccessible from Eden, limiting flexibility and leading to inconsistency if one changes DoNaLD state that is stored in EDEN.

EDEN however determines types of variables automatically. The massively increased CPU performance now available makes it more viable today to perform more of the definitive notation translation task in Eden, leading to flexibility gains and potentially less memory usage.

There is another motivation for performing definitive notation translation in Eden. The integration of notations into the single `tkeden` tool removed the shortcomings noted by Y.P. Yung (§4.1.5): it is possible for EDEN to generate DoNaLD or SCOUT definitions and cause them to be parsed using the `execute()` command, and the integrated system has led to an improved user interface, making the tool more accessible and allowing the research group to focus on applications. However it gave the integrated notations, SCOUT and DoNaLD, undue prominence, making it difficult to use other also interesting notations such as ARCA [Bey83], CADNO [Bey89b] and Sand (*sandSockett1992*). The integration also limited the possibilities for experimentation with creating and modifying definitive notations.

The preceding paragraphs, then, have given the motivation for implementing EDEN "in itself". The following sections describe progress related to this theme.

4.2.2 A database in EDEN: EDDI

In 1996, S.V. Truong created the EDDI (EDEN Database Definition Interpreter) notation [Tru96]. It is a definitive relational database language based on ISBL [Tod76]. The syntax is shown in Figure 4.15.

The syntax used to define views in EDDI (which, in Figure 4.15 includes the "define view" command and the syntax listed below it) can be considered to make up a pure definitive notation. It can thus be translated into Eden definitions. The procedural statements (above the "define view" command) can be translated into

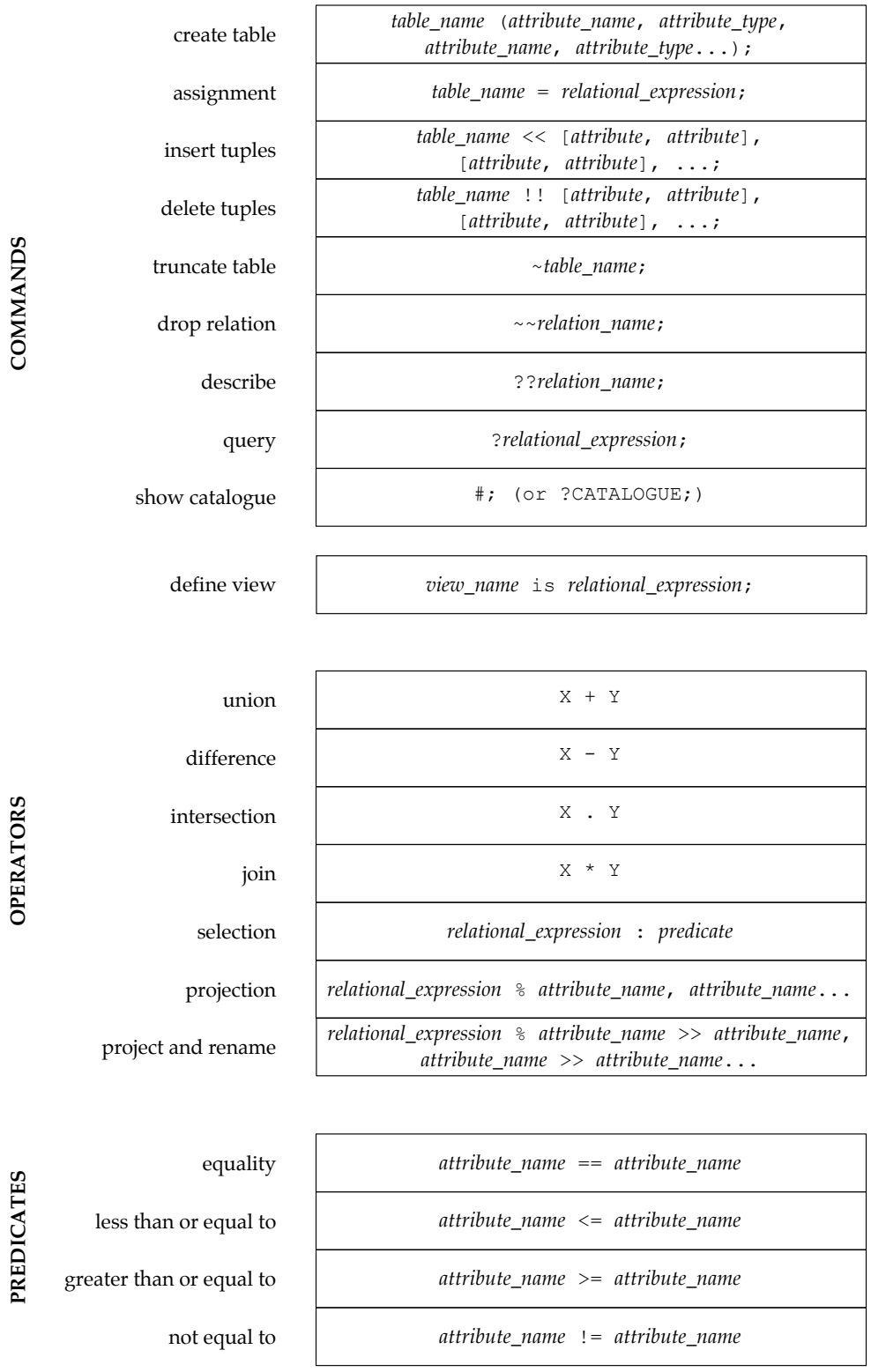


Figure 4.15: EDDI syntax

procedural Eden statements.

Truong created two implementations of EDDI based on `ttyeden`. EDDI/R (*eddirTruong1996*) was the 'real' implementation, where `ttyeden` was connected to an Oracle database by extending EDEN with functions written in Pro*C containing embedded SQL. EDDI/R has not been seriously used by anyone except Truong and is not discussed further here.

EDDI/P (*eddipTruong1996*) was a 'pseudo' implementation of EDDI. In EDDI/P, the relational data is actually stored as lists in the EDEN symbol table, rather than in an Oracle database, hence the 'pseudo' nomenclature. EDDI/P is a translator implemented as UNIX filter, designed to pipe into `ttyeden`, for example using the command:

```
cat eddipf.e - | eddip | ttyeden -n
```

The example given in Listing 4.9 shows EDDI/P in use, and also the (normally hidden) translator output. Textual formatting is here used to distinguish between `the EDDI input`, *`translator Eden output`* and the `output to the user`. Note that utility functions written in Eden (e.g. `create`, `project`, `inter`) are used in the translator output in much the same way that the DoNaLD translator uses EDEN procedures to create geometric objects and perform operations on them (*cf.* §4.1.4). In EDDI, these utility functions are defined in the `eddipf.e` Eden support library file loaded as a part of the UNIX pipeline above.

The EDDI/P translator together with `ttyeden` was provided by the author and Beynon as a student resource to assist with the teaching and learning of relational algebra in a database systems module at Warwick, originally in 1999 and again in 2000 (*eddipWard2000*).

4.2.3 A front-end parser in EDEN: the Agent Oriented Parser

The original EDDI system, being based on `ttyeden`, provided only terminal-based interaction with relational algebra. Using `tkeden` would provide an improved interface and it was thought that potentially the graphical facilities of `tkeden` could be used to provide visualisation of the database. However, it was not possible at that time to pipe input into `tkeden` and, because of the reasons stated above, it was desirable to avoid adding the EDDI/P translator code, written in C, to EDEN.

```
%eddi
FRUITS (NAME char key, BEGIN int, END int);
  FRUITS = create("NAME","#","BEGIN","","END","");
FRUITS << ["granny",8,10],["lemon",5,12],["kiwi",6,7],;
  FRUITS = addvals(FRUITS,["granny",8,10],
    ["lemon",5,12],["kiwi",6,7]);
?FRUITS;
  showrel(FRUITS);
-----------------------------------------------
NAME            BEGIN           END
-----------------------------------------------
granny          8               10
lemon           5               12
kiwi            6               7
-----------------------------------------------
NAMES is FRUITS % NAME;
  proc _rt2: FRUITS { _re2 = project(FRUITS,["NAME"]); };
  NAMES is _re2;
?NAMES;
  showrel(NAMES);
----------------
NAME
----------------
granny
lemon
kiwi
----------------
SUMMERFRUITS is ((FRUITS: BEGIN >= 6) . (FRUITS: END < 9)) % NAME;
  proc _rt5: FRUITS { _re5 = select(FRUITS,"BEGIN",">=",6); };
  proc _rt7: FRUITS { _re7 = select(FRUITS,"END","<",9); };
  proc _rt8: _re5,_re7 { _re8 = inter(_re5,_re7); };
  proc _rt9: _re8 { _re9 = project(_re8,["NAME"]); };
  SUMMERFRUITS is _re9;
?SUMMERFRUITS;
  showrel(SUMMERFRUITS);
----------------
NAME
----------------
kiwi
----------------
```

Listing 4.9: Interaction with the `eddip` translator

A method of replacing the `eddip` pipeline translator was devised; it makes use of what is now called the Agent-Oriented Parser (AOP). The AOP is written mostly in Eden (with a few small changes required in the EDEN C code to arrange for input to be passed to the Eden interpreter) and is capable of parsing many languages. More specifically, however, first we consider the AOP as a black box translator, parsing EDDI input. The upper part of Figure 4.16 shows the AOP translation of a line of EDDI input to several Eden definitions.

The AOP EDDI translator creates several lines of EDEN output for one line of EDDI input. The several Eden definitions that are created are each of a relatively simple nature. In this respect, the AOP EDDI parser resembles the DoNaLD to DAMscript compiler described in §3.3.2. Sub-expressions of the original input can be observed: for example, `var_60` in Figure 4.16 holds the value of the sub-expression `CITRUS % NAME`. This value could potentially be re-used if another definition required the same sub-expression, reducing the total amount of re-evaluation time. Parts of the original expression can also be modified without a need to invoke the parser again.

Although the mapping of a single EDDI definition to many Eden definitions described has these advantages, the result can be inconvenient when EDDI and Eden are being used in close conjunction in a complex EM exercise. In her final year project [Oun04], Asma Ounnas has written Eden procedures to recursively modify the script graph that results from an AOP EDDI translation, transforming the structure into a form that is isomorphic with that of the input. The modification is possible due to the simple "referentially transparent"[15] nature of a set of definitions. Structural changes such as these can modify the amount of internal detail in a script graph, whilst leaving the maintained relationships between leaves and roots unchanged. Although the post-processing technique shown in Figure 4.16 seems inefficient, it provides more development options for the modeller — the removal of internal detail may or may not be significant.

Thus far, the AOP has been discussed as a black box. The following is a brief overview of the internal operation of the AOP and its subsequent development and use. It illustrates a pattern of development through progressive construction by a

[15] I place the term in quotes as referential transparency in a definitive script applies only within an unchanging state.

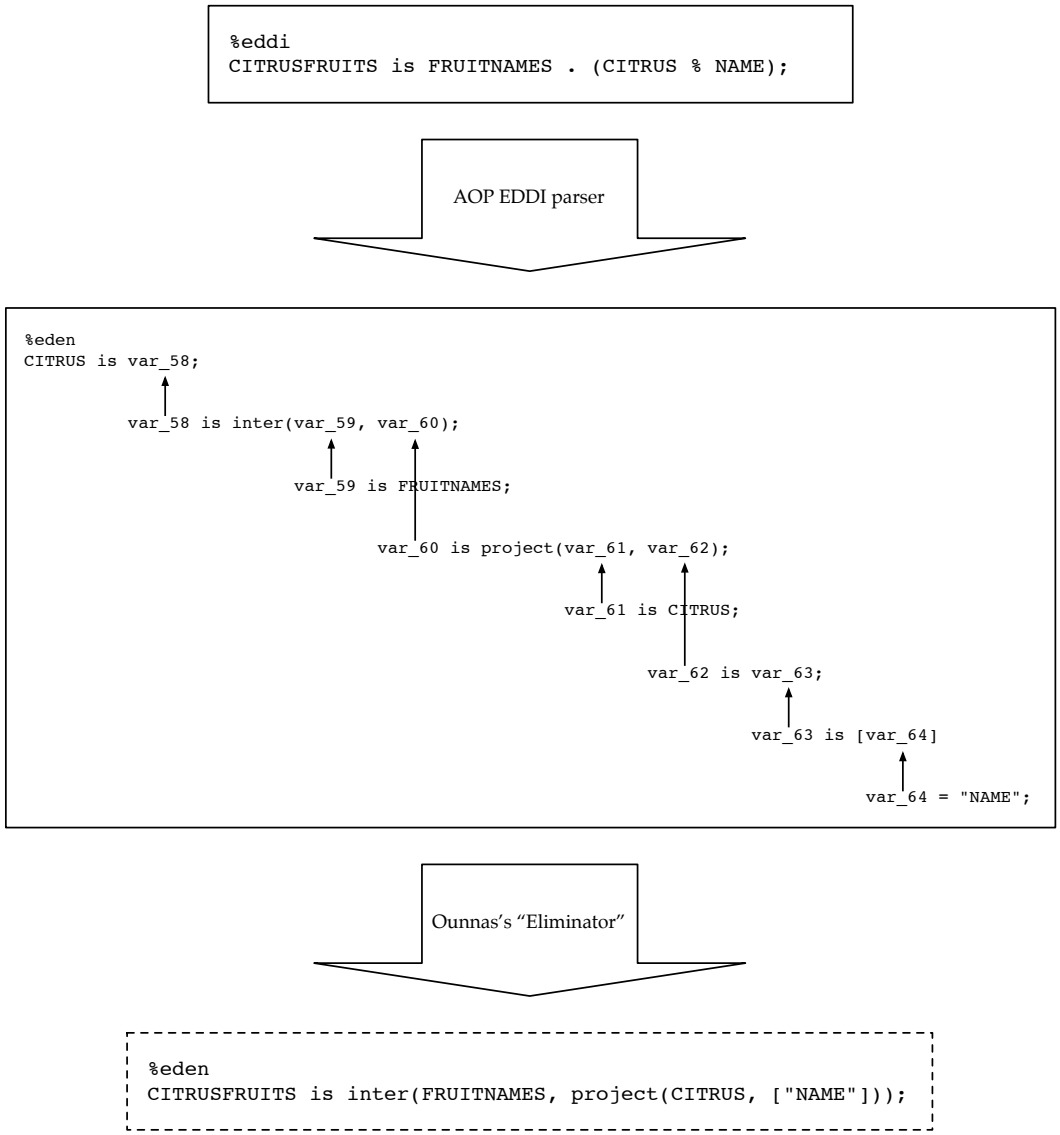


Figure 4.16: AOP EDDI to EDEN translation

number of varied authors which is not unusual in EDEN model development, to which my work has made a crucial contribution.

The AOP is originally the work of Brown (*agentparserBrown2001*). He wrote AOP configurations that enabled the parsing of several types of input, including palindromes, the procedural language PL/0[16] and EDDI. Brown's EDDI parser used the `eddipf.e` Eden support library file originally developed by Truong (*eddipTruong1996*). Evans and Brown created an SQL AOP parser configuration in September 2001, which I included in version 1.26 of the EDEN distribution. Beynon improved the SQL parser (*sqleddiBeynon2001*) and used it as the successor to EDDI/P when teaching the Warwick database systems module in November 2001. In February and March 2002, Roe produced a parser configuration and interactive environment for the LOGO language (*logoparserRoe2002*) and constructed a simple clown-control notation constructed on top of LOGO (*krustyRoe2002*). In August 2002, I added regular expression processing facilities to EDEN. Harfield simplified the AOP in May 2003 (*agentparserHarfield2003*), incorporating regular expressions to parse literals. He also documented the AOP and wrote parser configurations for a simple calculator notation similar to `hoc` (§4.1.1) and a parser configuration for a systolic array definitive notation (*sandHarfield2003*), originally developed in C++ with `lex` and `yacc` support by Sockett in 1992 (*sandSockett1992*). In July 2003, Roe and I developed a presentation for a workshop on Teaching, Learning and Assessment in Databases [BBRW03], using EDEN to generate PowerPoint-like slides, some containing live demonstrations of EDDI (see §4.2.6).

As an indication of the nature of the work that was involved in supporting the above development, in version 1.43, I generalised the notation handling framework (see §4.2.4) and added regular expression processing facilities to EDEN. The use of regular expressions is well documented in Frield's "Mastering Regular Expressions" [Fri97]. For implementation, rather than use an operating system library for regular expressions and risk incompatibilities between EDEN tools running on different platforms, I chose to use Hazel's "Perl-compatible regular expressions" package [Haz]. I used the resulting facility to construct an Eden preprocessor '`%edensl`' in an attempt to ease problems of using Eden lists definitively (see §5.2.1).

[16] Introduced by Wirth [Wir76].

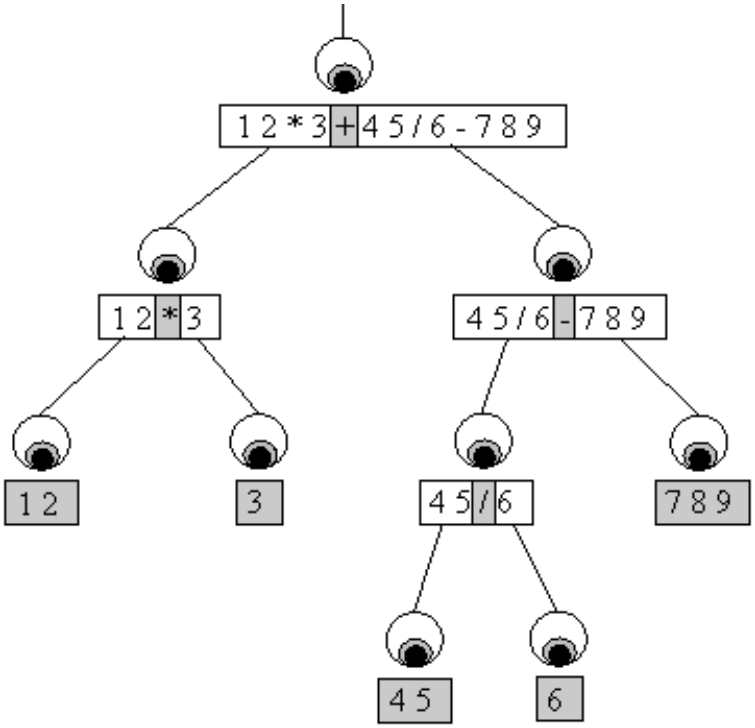


Figure 4.17: Observational structure of the AOP (from *agentparserHarfield2003/docs/tutorial/*)

The AOP is used to construct a parser whose operation can be understood using the EM concepts of observation, dependency and agency. Initially, an 'agent'[17] observes the entire input string in an attempt to locate the token considered most significant in the definitive notation being parsed. If the token is found, the agent creates a definition whose sources are initially undefined (`@`), and then instantiates child agents to generate definitions for the remaining sub-sequences of the input (see Figure 4.17). The method of repeatedly sub-dividing the input string leads to three interesting consequences:

- The termination of the parser is in principle guaranteed, since the primary activity involved is sub-division of the input string[18];

[17]This term is used as a reference to the ADM and LSD sense of the word but this sense is lost somewhat in the EDEN implementation.

[18]Operator precedence is currently handled by the automatic insertion of parentheses around the highest precedence operators and operands, and this can only extend the length of the input by a bounded amount.

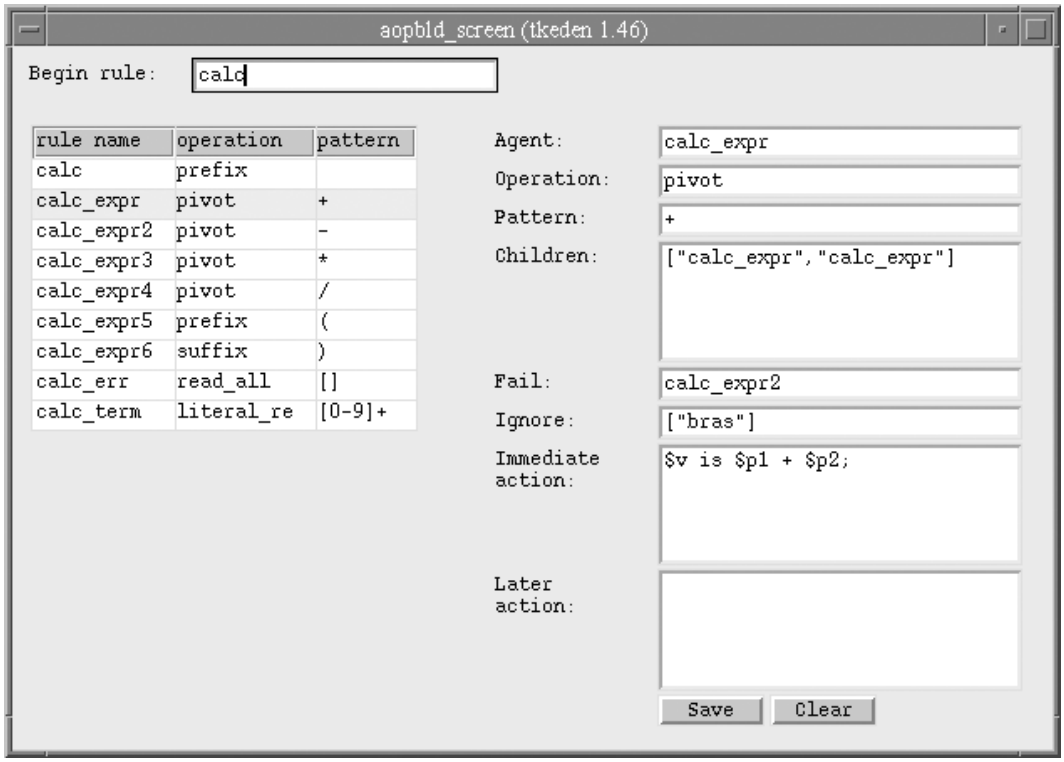


Figure 4.18: Harfield's parser builder in *agentparserHarfield2003*

- When a parse error occurs, the erroneous substring can be provided to the user for correction, rather than the entire input;

- Similar to definition maintenance (see §5.1), the child agents could in principle execute in parallel, or a depth- or breadth-first implementation can be chosen. The original author, Brown, experimented with depth- and breadth-first implementations — the current EDEN does not support concurrent execution.

The instantiation relationships between agents are described in an AOP parser configuration, where the notation grammar and parser agent actions are encoded into Eden lists. Harfield has built EDEN models using DoNaLD and SCOUT to show graphically the parser configuration and also how a particular input is parsed: see Figures 4.18 and 4.19 respectively.

The parser configuration can be changed interactively whilst the EDEN system is running. However this does not currently lead to a re-parsing of the input.

As the parser configuration is represented as Eden lists, it can be made dependent

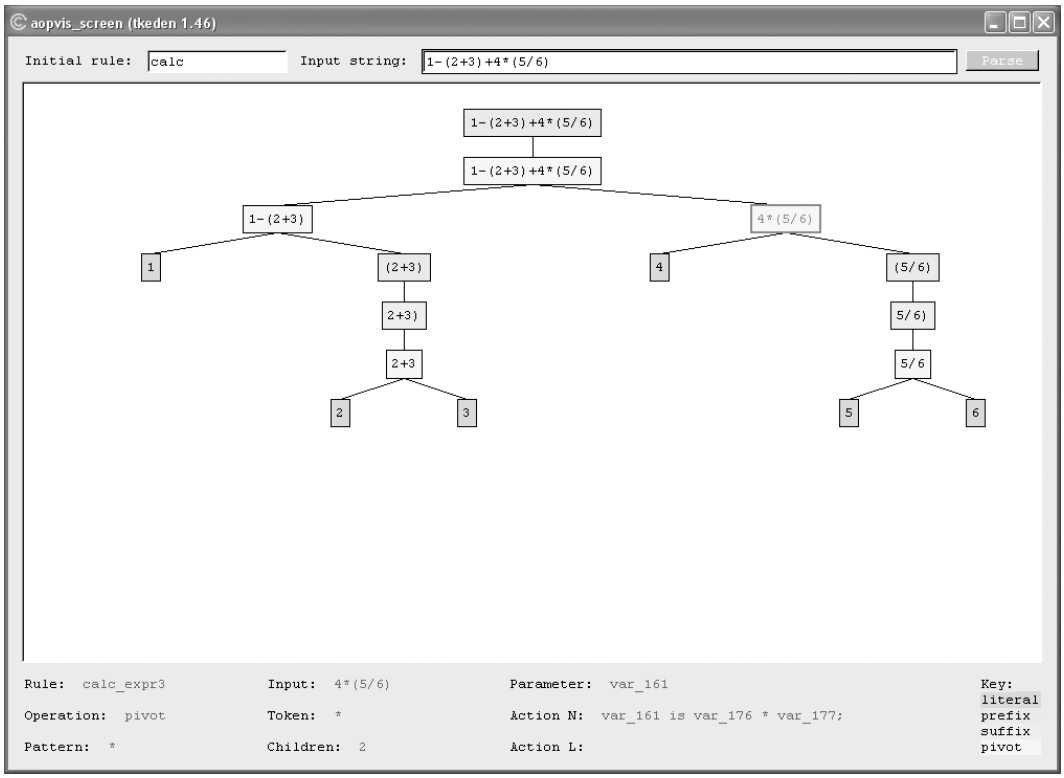


Figure 4.19: Harfield's parser visualisation in *agentparserHarfield2003*

Figure 4.20: `tkeden` input window with a selection of the multiple notations now available installed

on other variables. Brown uses this idea in his palindrome parser, where a parser configuration causes the prefix and suffix of the input to be repeatedly read and checked for equality. Dependency is used within the parser configuration in such a way that the suffix will always be checked with the last read prefix.

The available parser configurations are made visible in `tkeden` through facilities I added in version 1.43 to create the necessary radio buttons when a notation is installed: see Figure 4.20, which shows the tool with many notations installed and ready for use.

4.2.4 Generalising the parser solution: the notations framework

A further improvement implemented in version 1.43 of EDEN is the generalised notations framework. Notations need not be implemented by writing an AOP parser configuration: if this is more convenient, they can be implemented directly in Eden, which can call into functions written in C if required. Installing a new notation is done by in EDEN by calling an Eden procedure `newNotation`, which requires the following parameters:

```
newNotation(name, switchProcPtr, transProcPtr);
```

The first parameter provided here is the name of the notation implemented. The second parameter is a pointer to a `switchProc` procedure, which will be called (in a form of 'call-back') whenever the Notation Director (see Figure 4.6, p.215) detects that the input mode has switched to the notation `%name`, allowing initialisation of

the parser. The last parameter is a pointer to a `transProc` procedure which will be called once for every character read from the input. This procedure is expected to translate the input and output the according Eden code using calls to the EDEN `execute()` command. The name of the notation is provided when the `switchProc` and `transProc` procedures are called and so the procedures can be shared amongst multiple notations (as they are in AOP notations, for example).

Notations installed in this way take precedence over the built-in notations and so can replace the existing DoNaLD, SCOUT and Sasami translators (which are written in C with `lex/yacc` support) or even replace the default Eden notation. This feature should allow improvement of the basic notations in the tool without the need for such a project to involve recompilation of the tool or knowledge of the C source code.

Augmentation of the existing notations is also possible as aliases for the existing notation names can be introduced. For example, the existing built-in notation `%donald` has an alias[19] `%donald0`. A new notation named `%donald` can therefore be introduced using `newNotation()`, and the implementation can generate `%donald0` output for cases that it does not wish to handle.

Notations generating output in notations other than Eden leads to the need for each parser to be re-entrant. This is an issue if, for example, a notation `%one` that is implemented using `switchProc1` and `transProc1` generates output which calls a notation `%two` that is implemented using the same procedures. The KRUSTY notation (*krustyRoe2002*) is dependent upon the LOGO notation (*logoparserRoe2002*) in this way as they are both implemented using the AOP. Work on the system to implement a re-entrant parser is near completion.

4.2.5 Making the parser dependency driven?

Given that the AOP is written in Eden, it may be possible to control the parser using dependency. That is, to make the Eden output from the parser dependent upon the input text to the parser. As described in §4.1.4, definitive notations are

[19] In version 1.43 and later — DoNaLD only at present.

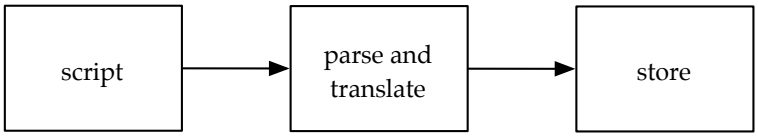


Figure 4.21: Notation input data flow in EDEN

generally implemented in EDEN by construction of a translator parser that converts definitive notation script input into Eden language output, which is then executed or stored by EDEN as appropriate. The data flow is shown in Figure 4.21. Presently, the data flow illustrated in Figure 4.21 is controlled in a conventional way: the entry of script input causes a procedure call to the translator with the portion of script passed as a parameter. The translator then uses further procedure calls to output Eden code and change the underlying state of the EDEN machine. The procedure calls form a 'one-off' process and little information about the process is retained after it has completed. Pursuing the "in itself" motivation, it would seem possible to construct dependencies for the edges in Figure 4.21, controlling the translator using dependency.

A dependency-driven parser would require that the EDEN system store all currently valid input (the 'script' input) in its original form[20], to provide input to the parser dependency. The dependency structure constructed would allow Eden code in the system to be traced back to the corresponding original input. (In the current EDEN, this is not possible without knowledge of translator conventions.) Changing the input text would cause it to be automatically re-parsed. Further, changing the parser itself would also cause the input to be automatically re-parsed, maintaining and guaranteeing the definitive relationship described by Figure 4.21.

Figure 4.21 actually describes a many-to-many mapping, since the script and store are both non-atomic areas of memory. Given a foundation capable of maintaining such a definitive mapping, it would be possible to make *sections* of the store dependent on *sections* of the script, in such a way that (for example) changing one character of the input would lead to the minimum necessary re-parsing.

[20]The input is in fact stored by the current EDEN system, but in a somewhat superfluous way: the only purpose it serves is for user observation in the interface (see the Eden Definitions window in Figure 4.5 on p.215).

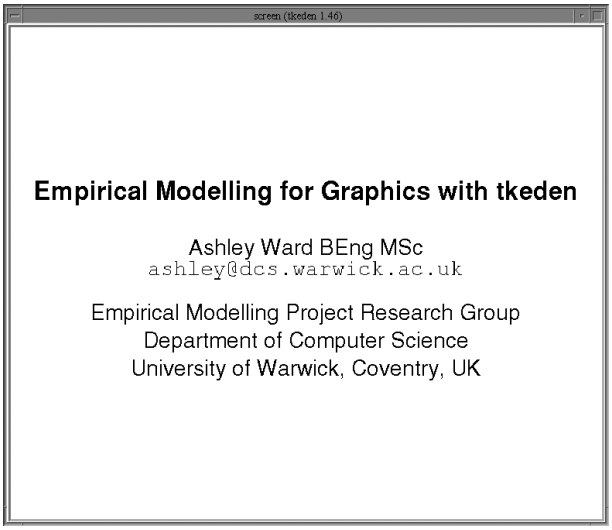


Figure 4.22: The first presentation environment slide

It is also interesting to note that the processing of the Eden language itself can also be described by Figure 4.21 — the EDEN implementation is a translator parser that generates EDEN VM code (see §4.3.5). Controlling this translation with dependency would be a further challenge.

4.2.6 Demonstrating EDEN with presentations

Prior to 2001, demonstrations of the EDEN tool usually took the form of an introductory talk supported by OHP or PowerPoint slides, followed by a short 'case study', showing EDEN and one or two models. Inspired by a demonstration of "Imagine" [KB00] (an OO LOGO tool), I created a basic presentation environment for EDEN, using it first for my presentation at the Libre Software Movement (LSM) conference, Bordeux in 2001 (*lsmpresentationWard2001*). My initial slide is shown in Figure 4.22.

I attempted to make it possible to use dependency as much as possible in the presentation environment. To enable this, I extended and improved the underlying EDEN tool in various ways. For example, the text shown in the slide happens to all be centred horizontally across the window. If the window is resized, the text repositions in order to remain centred in the window. The text repositioning was achieved by extending EDEN to include a `screen_width` variable and by rewriting

```
titlef = "{helvetica 27 bold}";

s101 = "Empirical Modelling for Graphics with tkeden";
s101f is titlef;
s101w is StringWidth("screen", s101f, s101);
s101x is (screen_width - s101w) / 2;
s101y is 200;
```

Listing 4.10: EDEN script corresponding to part of Figure 4.22

an existing `StringWidth` function (initially written in C) in Eden. These two features allow dependencies in the form shown in Listing 4.10 to be created. Changes to the text, font or window width then propagate and cause repositioning of the on screen text.

The `x` coordinates of the text are therefore calculated by dependency in order to centre the text on the screen. The `y` coordinate of the first line of text on the screen is given using a literal value, but subsequent lines of text are positioned with reference to the height and position of lines of text above, again using dependency in a similar way.

This first slide has six items of text, some with different fonts and line spacing. In an attempt to make slide editing easier, I used Y.W. Yung's `macro` function [Yun90, Appendix B] to perform pre-processor-like substitution on a script template, which is then instantiated using the Eden `execute()` function. This is analogous to an instantiation of an object from an OO class template, using a constructor. I wrapped this in an Eden "string, centered" procedure named `sc()`. The corresponding EDEN script that configures the slide is shown in Listing 4.11.

As well as text, a slide in the presentation environment can also contain an existing Eden model. I included the 'jugs' model (*jugsBeynon1988*, [BY90]) in a slide, and also displayed part of the definitive script for jugs beneath the model itself, showing the current value of definitions — see Figure 4.23. Again, this display is created using dependency as much as possible. As a result, changes to the state of the jugs model, made by pressing the Fill/Empty/Pour buttons or by redefining the script, appear in the slide's display of the definitive script. This was an attempt to make it easier for an audience to comprehend the indivisible linkage between the

```
sc(101, "Empirical Modelling for Graphics with tkeden", "titlef",
     "200");
sc(102, "Ashley Ward BEng MSc", "plainf", "s101y + s101h*2");
sc(103, "ashley@dcs.warwick.ac.uk", "codef", "s102y + s102h");
sc(104, "Empirical Modelling Project Research Group", "plainf",
     "s103y + s103h*2");
sc(105, "Department of Computer Science", "plainf",
     "s104y + s104h");
sc(106, "University of Warwick, Coventry, UK", "plainf",
     "s105y + s105h");

%scout
display page1 = <s101win/s102win/s103win/s104win/s105win/s106win>;
```

Listing 4.11: EDEN script that instantiates the slide shown in Figure 4.22

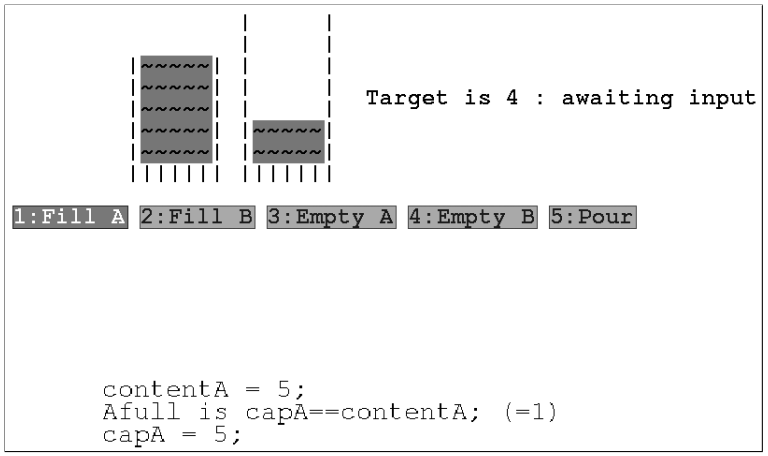


Figure 4.23: The 'jugs' model in a presentation slide

conventional-looking interface rendering of jugs and the unconventional, previously invisible, internal script state.

Models included on a slide in a presentation continue to 'run' when the slide is not visible, as EDEN maintains all current definitions in the system. Thought of in operating system (OS) terms, then, EDEN allows multiple models to be loaded[21] and to "execute concurrently". (Of course, EDEN does not provide many of the usual features of an OS and presently requires a conventional OS in which to function — but despite this, the analogy is interesting.) Unusually for an operating system, the data of executing 'processes' can be indivisibly coupled in un-preconceived ways. For example, the level of water in the left jug (`contentA`) can be linked to the `y` coordinate of the text on a particular slide, or to the size of a cube in the Rubik's model (see earlier §4.2.1). The "inter-process communication" is handled by the EDEN definition maintainer.

4.2.7 Interface improvements

I have attempted to incrementally improve the `tkeden` interface at many points throughout the fifty versions. Figure 4.24 compares the interface from 1997 (when the tool ran only on one platform) to version 1.46 (October 2002). The menus available in the newer version are shown in Figure 4.25. In this figure, the new Edit and Help menus are the primary change from the 1997 interface.

Visible improvements in the user interface include new Help menus, containing condensed information on each notation. After observing many people make redefinitions in the wrong notation context, I added the radio buttons above the input window to make the current notation context more explicit and easier to change. To enhance the input window, I changed the background colour to white, improving readability both on screen and in printed screenshots, enlarged it slightly to show more context and improved the speed at which the cursor can be located by colouring it and making it flash. Other improvements include more keyboard shortcuts and more uniform use of common conventions — for example, using the ellipsis to mark items requiring further input in menus and showing the 'Accept' action as the usual action-implying button rather than choice-implying menu.

[21] Providing their variable identifiers do not clash — EDEN has only a global namespace.

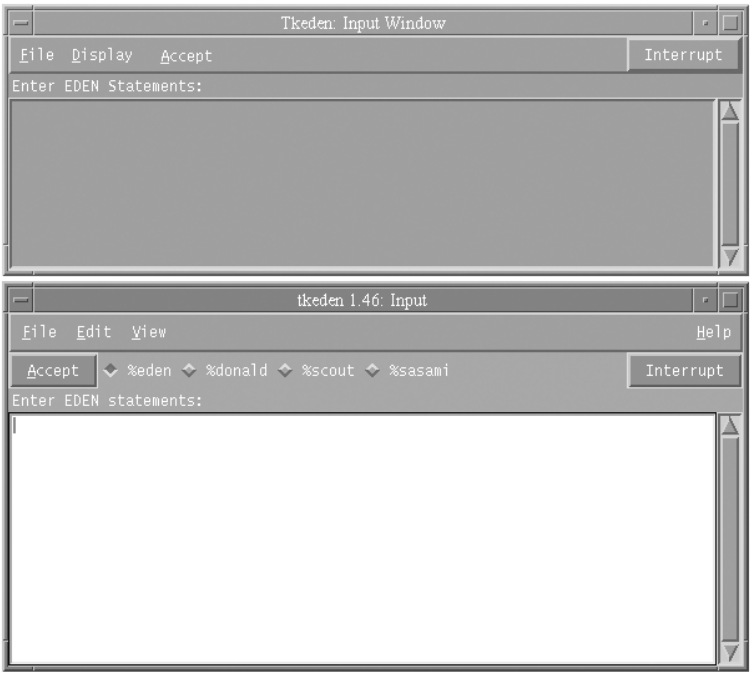


Figure 4.24: `tkeden` interface changes: `tkeden-dec151997` contrasted with `tkeden-1.46`

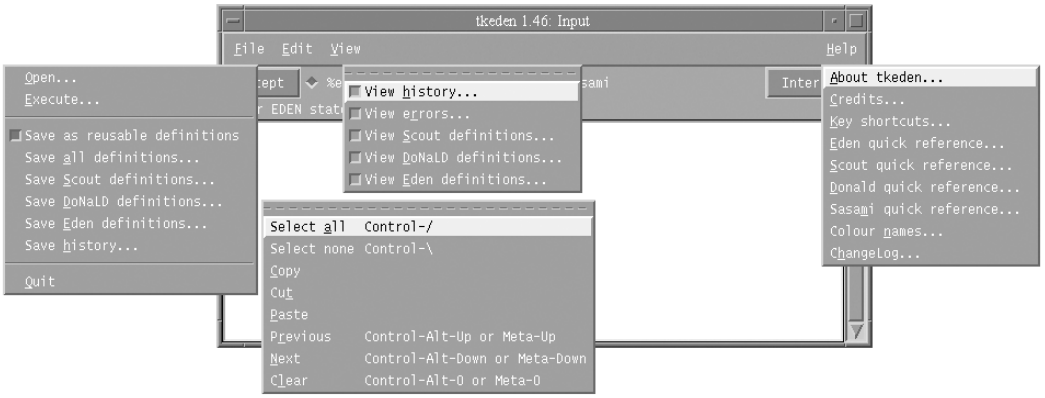


Figure 4.25: `tkeden-1.46` menus

In addition, I improved the error messages. This work frequently went well beyond improving the text of each message, involving new implementation to present error information meaningful in the users' terms and the context for the error.

Although these improvements may seem trivial, where possible they are motivated by theory (e.g. the principles outlined in [app87]) and they do seem to have enhanced the usability of the tool and general acceptance to the audience — who are primarily undergraduate students using the tool for a relatively short amount of time. In addition, frequent releases of the tool have helped to generate feedback to steer the development. There is much scope for further improvement, particularly regarding input methods other than the input window — for example, editing the script in place, and mouse-driven creation of DoNaLD and SCOUT definitions. Wong [Won03, §7.3.2] includes an evaluation of the `tkeden` interface and includes further systematic treatment of interface issues in definitive environments.

One use of the presentation environment outlined in the previous section §4.2.6 shows a possible future direction for implementation of the EDEN interface. For a workshop presentation in 2003, Roe implemented a SCOUT model of a simplified version of the `tkeden` interface (which is currently implemented in Tcl/Tk), allowing integration of the interface into a presentation slide, as shown in Figure 4.26. Implementation of the entire interface itself in SCOUT (following the "in itself" theme) would allow use of dependency in implementing the interface, reduce the reliance on the present Tcl/Tk foundation and encourage tool users to modify the interface for their needs. A prerequisite here however would be various improvements to SCOUT, perhaps including the ability to use platform-native widgets.

The remainder of this section outlines another key aspect of improvements to the interface — the use of redefinition history.

Much of the potential attractiveness of a definitive tool comes from the ability to interactively make redefinitions to a model. If the interface does not support the modeller in easily and efficiently identifying and then making a series of redefinitions, then much of the power of the concept is in practice hidden from the user.

A key realisation that has guided some of my improvements to the EDEN interfaces is the common need (when undertaking exploratory modelling) to make a *series* of *similar* redefinitions. Often, feedback from making a redefinition leads to

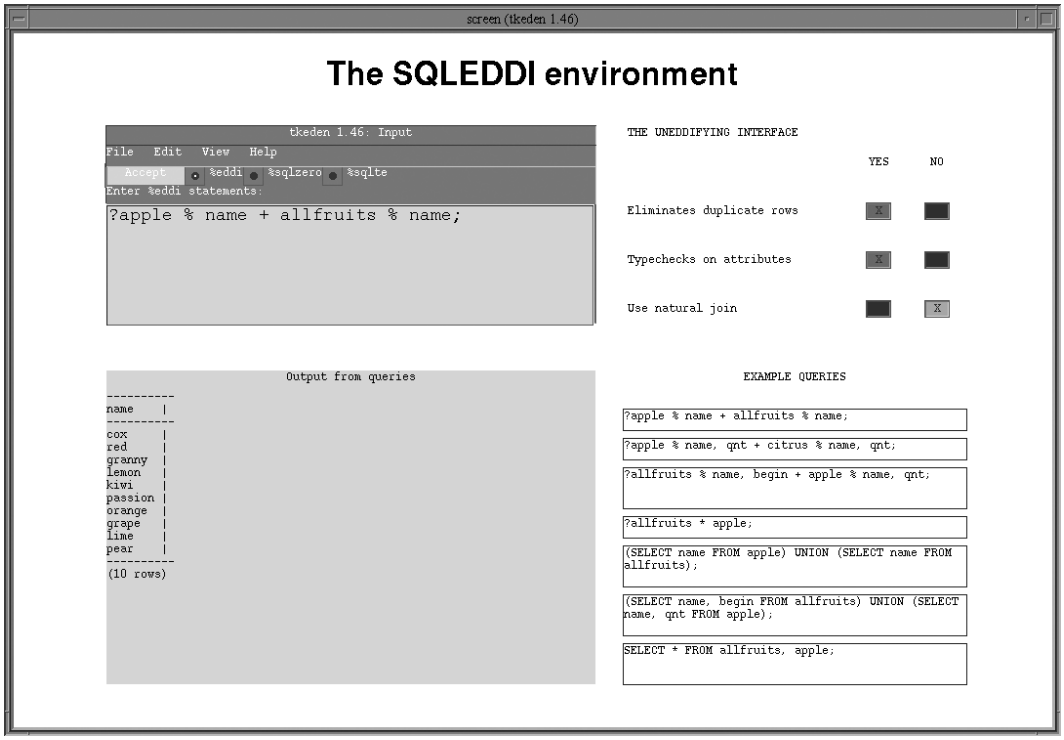


Figure 4.26: Part of a simplified `tkeden` interface constructed in SCOUT

the realisation that it can be improved in some way, and so it is desirable to be able to edit it. The original EDEN interfaces did not easily support this kind of activity.

In early versions of `ttyeden`, I took advantage of the GNU readline library [Ram] to implement cursor-driven line editing of the current input, including the ability to recall and edit previous input — the same facilities that are routinely provided in a UNIX shell. Later[22] I implemented a similar facility in `tkeden`.

Users report that these additions have improved the usability of the interface greatly, and one hopes that the underlying concepts used in the tool are clearer as a result. However, these particular implementations are not quite satisfactory as they deal with only the textual user input and history thereof. There is a great potential for integration with the underlying machine state. First, a relatively trivial example is the ability to assist the user in making redefinitions, based on current state. The ability to automatically complete the typing of references to variables — similar to how in many UNIX shells, references to filenames can be completed by pressing the TAB key — would improve accuracy and speed of redefinition input.

More significantly, use of the *history* of the underlying machine state could lead to improvements that again would reveal more of the power of the definitive concept. It should in principle be possible to wind the machine state backwards, 'undo'-ing through the history of the modelling session. Roe [Roe03, §2.3.1] reviews a similar facility in the Forms/3 tool [BAD+01].

The current tool saves the history sequence of interaction during a session to a file[23]. The file also shows any error messages received, allowing the reader to distinguish successful and unsuccessful lines of input. I have made it possible to reinterpret this file without modification in most circumstances by introducing a new one-line comment syntax available in all notations with which to mark the errors, using the `##` character sequence to mark such a comment line. (The single `#` character could not be used for this purpose as it already denotes the list length operator in Eden.)

In his final year project [Kin04], Karl King has recently implemented a limited 'checkpoint' facility in `tkeden`, where the current state can be marked and later

[22] In version 1.13.

[23] Here, the sequencing of the lines in the file are usually significant, so pedantically this should not be called a 'script'.

returned to. More generally, it should be possible to move backwards through the history sequence, and then move forward from a particular point, creating a history *tree* of redefinitions, the branches of which could then be later recombined in different ways. There are many precedents for this idea — for example, the RCS system [Tic85] and the later CVS system [cvs] allow 'branches' of source code history to be created and later merged, and the music production tool 'Digital Performer' allows branches to be created within audio editing history [mot]. Definitive state seems to be well-suited to such experimentation as referential transparency means the context needed for a definition is clear.

There are semantic problems with histories of redefinitions, however. For example: should the loading of a file of definitions be recorded in the history with a reference to the file identity or to the file contents? If the system makes a redefinition when the mouse is clicked, should this kind of redefinition be recorded in the history? Because of the conflation of tool and model, modelling and use, program and data in the history, there are no absolute answers to these questions — a problem which is exacerbated by an increasing amount of the tool being implemented "in itself".

4.2.8 Novel interfacing

To conclude this overview of fifty versions of EDEN, I give three examples of recent novel usage of the tool, each involving interfacing EDEN to other systems through various extensions. The first two examples involve interaction with hardware devices, and the third interaction with external software. All three examples show various problems of concurrent synchronisation.

The Universal Serial Bus (USB) is an interface standard introduced in 1996 that allows connection of peripheral devices to personal computers. Devices are categorised into device classes which define generic behaviour and protocols. One such class is the Human Interface Device (HID) class. The USB and HID specifications are managed by an industry consortium and specifications can be downloaded from [usb]. Devices in the USB HID class include keyboards, mice, joysticks, data gloves, peripherals used for control of computer games (e.g. steering wheels, throttles, rudder pedals) and also devices that do not require human interaction but provide data

```
%eden

wheelfd = usbhidopen("/dev/input/event0");

proc readwheel {
  auto l;
  l = usbhidread(wheelfd);
  if (l != []) wheel = l;
  todo("readwheel();");
}

readwheel();
```

Listing 4.12: Using the USB HID facilities in `tkeden`

in a similar format to HID class devices, for example thermometers.

I introduced a capability for communication with USB HID devices in version 1.48 of `tkeden` on Linux, writing C code that builds on the support available on that platform to implement 'built-in' Eden routines `usbhidopen`, `usbhidread` and `usbhidclose`. The Eden code in Listing 4.12 uses this facility to continually read input from a USB steering wheel that is mapped to the Linux device `/dev/input/event0` and sets the Eden list variable `wheel` to the state read. The Eden `todo()` procedure is used to cause reading from the device to be interleaved with user input in a continuous loop.

After the code in Listing 4.12 has been introduced, the Eden variable `wheel` contains information read from the steering wheel. When the wheel is moved, the variable is changed by the `readwheel` procedure. The value can then be used definitively. The fourth item in the `wheel` list contains the angular position of the steering wheel. This can be connected to the position of the Sasami axes by introducing the following single Eden definition '`sasami_camera_rot is [0, 0, wheel[4]];`'. The introduction of this single definition causes the Sasami window to rotate in response to a move of the steering wheel, as shown in Figure 4.27.

Jon McHale, a third year undergraduate project student, built on this facility, constructing a `dtkeden` environment to be used for simulating reverse car parking (*carparkingsimMcHale2003*). The environment can be seen in use in Figure 4.28.

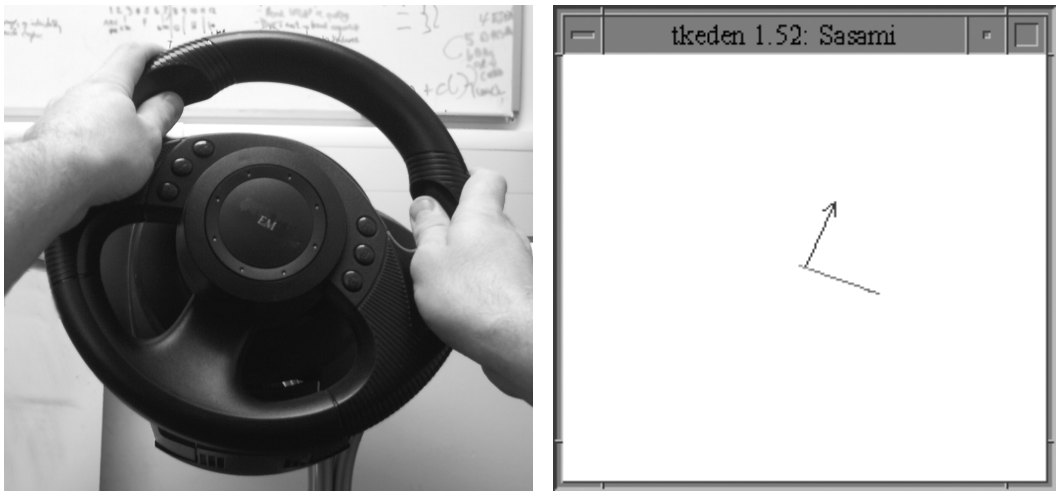


Figure 4.27: Sasami window linked to a USB steering wheel using a definition

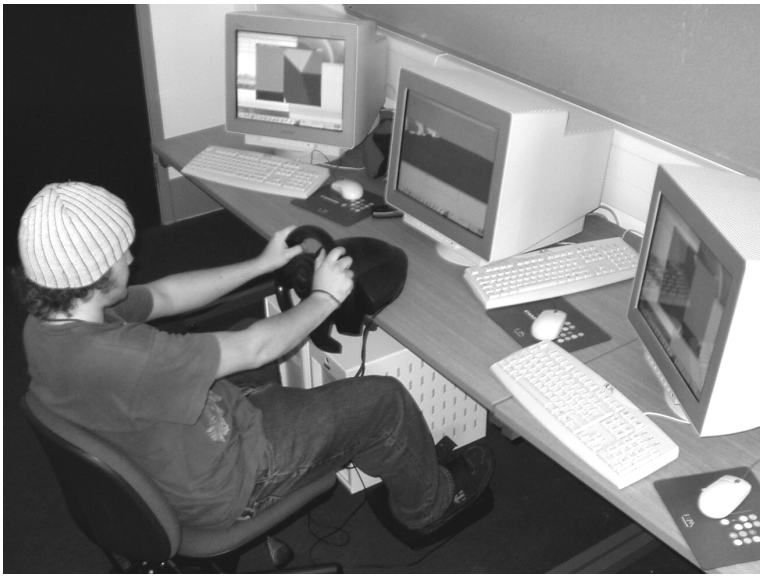


Figure 4.28: Jon McHale and *carparkingsimMcHale2003*

McHale used four machines to display four views from the simulated car: front view, left and right wing mirrors and rear view (not shown in the figure). The simulation was constructed using the Eden, SCOUT, DoNaLD and Sasami notations — a SCOUT interface being used to configure an Eden model of the physics moving a two-dimensional DoNaLD rectangle 'car' within a 2D environment (the DoNaLD `intersects` function being used to check for collisions), which was linked by dependency to the 3D Sasami represention running on multiple workstations on a LAN — the distributed communication of redefinitions being a feature of `dtkeden`.

The second example of novel interfacing involves output to as well as input from hardware. Rod Moore and Stuart Valentine (department technicians) constructed a small OO-gauge railway based on the Leamington to Birmingham connection and a system based around a PIC16F877 microcontroller which controls the power applied to the points and the rails (the power to the rails is pulse-width modulated to give control of locomotive speed). The PIC also reads from a set of twenty reed switch sensors positioned around the track. Each reed switch is closed when a magnet attached to the locomotive passes by, allowing the location of the train to be determined at that moment in time. The PIC microcontroller communicates via a serial port, sending a packet of information containing the state of the set of reed switches when a change is detected. It can receive simple commands to apply power to the rails and change the state of the points through the same serial port. The system is shown in Figure 4.29.

Using techniques similar to those described above for USB HID, I added code to EDEN to allow communication with devices through an RS-232 serial port on Linux, adding `rawserialopen`, `rawserialread` and `rawserialclose` functions accessible from Eden. I then wrote Eden procedures as outlined in Listing 4.13 to communicate with the PIC microcontroller. These procedures cause redefinitions to the Eden variable '`trainMOTION`' (for example, to the value "`f7`", meaning "Forward" at speed "7") to be propagated to the PIC microcontroller, and also the Eden variable '`sensors`' to be redefined when the train passes a location sensor. The lower portion of the example in Listing 4.13 shows how, for example, the direction of travel of the train can be reversed whenever a signal is detected from a location sensor, demonstrating input and output.

Figure 4.29: Chris Rose and the model railway hardware

```
fd = rawserialopen("/dev/ttyS0");

proc sendTrainCommand : trainMOTION {
  rawwrite(fd, trainMOTION // "\r");
}

proc readFromTrain {
  c = rawserialread(fd, 1);
  /* omitted code: ... parse input read one character at a time
     from rawserialread.  If input exists, redefine the
     'sensors' variable with the new state... */
  todo("readFromTrain();");
}

/* ... now, for example: */
direction = 1;

proc reverseOnPassingSensor : sensors {
  direction = !direction;
  trainMotion = direction ? "f7" : "r7";
}
```

Listing 4.13: Eden code to communicate with the train PIC

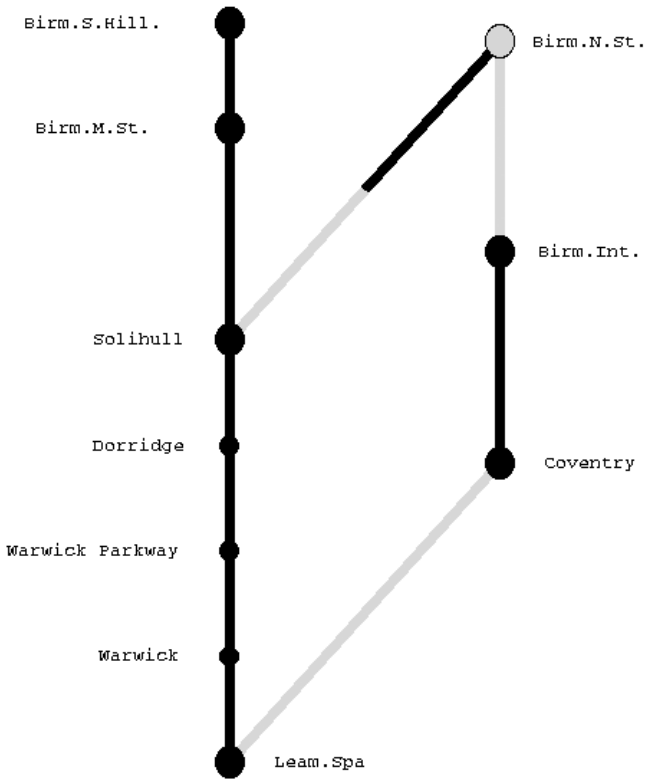


Figure 4.30: A partial screenshot of *modelrailwayRose2004*

Chris Rose, a third year undergraduate student, modified an existing `tkeden` model of a railway (*railwayYung1995*) to match the connectivity of the hardware model. The resulting model contained a schematic representation of the track layout constructed in DoNaLD, where the last received location of the train was indicated by colouring of a 'train indicator' node in the diagram — this being achieved by dependency on the `sensors` variable. The `tkeden` model (*modelrailwayRose2004*) is shown in Figure 4.30.

The original railway model (*railwayYung1995*) was a simulation whose progress was driven by a clocking procedure that is standard in many Eden simulation models. A simple version is below.

```
proc increaseClock : clock {
  todo("clock = clock + 1;");
}
```

Here, the `clock` variable is incremented as fast as the model can execute. (The semantics of the `todo()` procedure are further investigated in the next section §4.3.) The procedure can be modified to take account of real time, where the clock variable is not incremented until after (at least) the necessary amount of real time has passed. (I wrote a procedure to this effect for the car parking model, where it was desired to refresh the 3D graphical displays at a given real-time frequency.) In this type of 'real-time' solution, if the model re-evaluation takes a longer amount of time than there is available (a case which cannot be detected in the current EDEN), then the model will start to run more slowly than it should.

In the `tkeden` railway model that uses real hardware, progress of the `tkeden` model is driven by movement of the hardware, not a clocked simulation. The above `increaseClock` procedure is therefore replaced, and the `tkeden` model becomes in effect an extension of the hardware. Changes in the `tkeden` model are driven by changes in the hardware, and *vice versa*. There is no control flow or visible event loop in the `tkeden` model, just a set of definitions, some of which are dependent upon inputs from hardware. Interestingly, if the 'train indicator' definitions are present in the model, then they will respond to movement of the train, even whilst the remainder of the model is being interactively developed. This can be thought of as analogous to a prototyped section of circuit that is responding to inputs concurrently with another section of circuit that we are presently developing. In this analogy, stopping `tkeden` corresponds to removing power from the circuit: the current state is lost.

The final example in this section (before we generalise to some relevant issues from each of these examples) is motivated by the existence of the 'pipeline' notation translators mentioned in §4.1.4. Several translators, generally written in C with `lex/yacc` support, exist for definitive notations that are not integrated with the current EDEN system: for example, the ARCA to Eden translator written by Stuart Bird in 1991 (*arcaBird1991*). If it is desired to use these various notations, then the translator can be made to read a prepared file and write to a file that is then read into EDEN, but this is not a very satisfactory procedure when it is desired to interact with EDEN through the translator.

The generalised notations framework (see §4.2.4) would allow a notation managed by an external translator process to be used in the same way as other built-in notations (e.g. a radio button can be used to switch to and from that notation context), if Eden could be used to communicate with the external translator process. There are several technical difficulties here, however.

1. If the standard I/O library of the external translator detects that it is not connected to a terminal, it buffers input and output in blocks, as this makes file access more efficient. Therefore if it is connected to EDEN via a standard pipe, it cannot be used interactively.

2. The external translator may generate errors and these should be presented in the same way as other errors in EDEN.

3. Reading from the external translator cannot be allowed to block EDEN if there is no output from the translator.

I solved problem (1) by connecting a 'pseudo-terminal' between EDEN and the external process, following the example in [Cur96, Appendix D]. From the viewpoint of the standard I/O library of the external translator, it appears to be connected to a terminal and hence uses minimal buffering. Problem (2) was solved by connecting EDEN to the standard error output of the external translator using a pipe. If error output is detected in this pipe, the Eden `error()` procedure can be called, which causes the error to be handled in the same way as other EDEN errors. The necessary communication graph is shown in Figure 4.31. Problem (3) was solved by adding a function to EDEN wrapping the UNIX `select()` primitive, allowing a check to be made for input before a potentially blocking read is made.

I call the result an "Interactive Process Translator" facility. An external notation translator can be interactively introduced into the EDEN system by typing, for example:

```
%eden
installIPTrans("%arca", "/dcs/emp/empublic/bin/arca.trans");
```

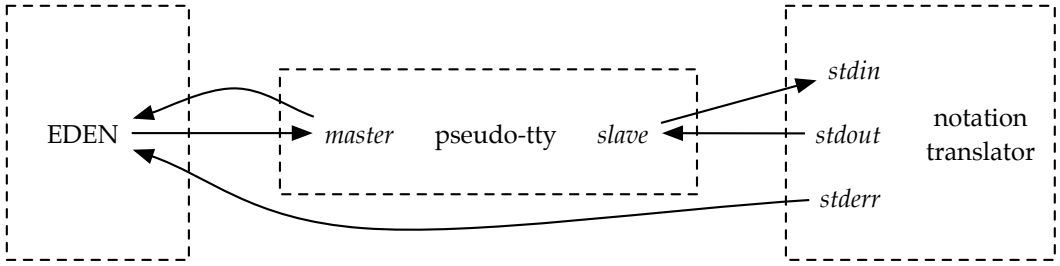


Figure 4.31: Communication between EDEN and an Interactive Process notation translator

Each of three above examples requires EDEN to read from a process or device which generates output asynchronously. The present solution in each case is to write an Eden procedure of the form:

```
proc readAsynchronous {
  if there is something to read {
    read the input
    deal with the input
  }
  todo("readAsynchronous();");
}

/* start the checking loop off... */
readAsynchronous();
```

The call to the Eden `todo()` procedure here causes the asynchronous input to be polled repeatedly. There are two problems. Firstly, if operating system mechanisms could be used by EDEN to cause the reading to be synchronised with the availability of some data to read, the inefficient polling would not be necessary. Secondly, the type of polling manifested by the `todo()` solution above is paused when EDEN is executing a sequence of statements, for example a `for` loop. User Eden code can therefore cause a delayed or missed response (depending on whether the input is buffered or not).

Both these problems emerge from the fact that the current EDEN is a sequentially executing machine, executing in only one operating system thread. The `todo()` procedure used here gives a partial solution, but a better solution requires concurrent execution within EDEN. The `todo()` procedure is investigated in more detail in the next section §4.3.

4.3 EDEN execution and scheduling

In practice, EDEN has proved to be by far the most effective 'general purpose' EM tool. It combines both dependency and agency in a way that we seem to find more acceptable than the ADM and the DAM machine, even making allowance for the relative immaturity of their implementations. Exactly how it works is therefore worth some investigation.

Y.W. Yung, who wrote the low-level machine algorithms that are still in use, starts his section on implementation [Yun90, §5] with a 'rule of thumb'.

> There are no strict rules for the behaviour of a definition maintainer. The evaluation of definitions can be eager, or lazy. The automatic re-evaluation can be enabled, or disabled, data driven or demand driven. No matter how the definition maintainer internally functions, it must ensure the values of formula variables are up-to-date. Perhaps it is the only rule for the definition manager to follow. I call it the "*Definition Manager's Rule of Thumb*".
>
> Definition Manager's Rule of Thumb: *Always evaluate a formula when a definition's value is referenced (if the value is not up-to-date).*

Here, Y.W. Yung is stating evaluation/storage strategy 3 (evaluate-at-use-when-out-of-date), which sits somewhere between strategy 1 (evaluate-at-use) and strategy 2 (evaluate-at-redefinition). This strategy requires formulae, values and out-of-date flags to be stored — it uses the maximum storage of the three strategies, with the aim of minimising the amount of evaluation. It is the most complex of the three strategies to implement and describe, which is why this chapter on EDEN appears after those on ADM and DAM.

Y.W. Yung does not state how the current EDEN maintainer operates. He does give an algorithm, but this describes the same depth-first evaluation scheme given in his original project report [Yun87], which was used only in very early versions of EDEN for which the source code is no longer electronically available[24]. The current EDEN uses a breadth-first evaluation scheme, about which Y.W. Yung gives only an example trace and a few short paragraphs [Yun90, §5.5.2]. The aim of this section is to describe the algorithm used and its consequences. This will be done by examining the artefact that remains — the tool — firstly as a black box, and secondly by examining the source code.

[24] Although a printed copy exists in Y.W. Yung's project report.

4.3.1 Terminology mostly explicitly dismissed

We have already briefly described the difference between Eden's formula variables and actions in §4.1.2. Now that we are examining the implementation more closely, it is necessary to look more closely at the distinctions made.

Y.W. Yung [Yun90, pp.4–5] makes a distinction between 'implicit' and 'explicit' actions:

> A spreadsheet is a program that provides us with a large grid of cells into which we can insert numbers and formulae. A cell that contains a formula causes the system to re-calculate the formula whenever any cell on which it depends is changed, and display the new value on the screen automatically. The display action is implicitly invoked by the system after the re-calculation. These actions can be called *implicit actions* because they are pre-defined in the system.
>
> The `make` command in UNIX is a good example of file maintenance software. Unlike the spreadsheet software, the user has to specify the updating action in a file (the `makefile`) explicitly. These actions can be called *explicit actions*.

This terminology is then used in the statement of the primary goals for the Eden language [Yun90, p.26]:

> 1. to support general purpose programming (e.g. in order to implement definitive notations)
> 2. to support definitions
> 3. to support explicit actions (since they are extensible and much more powerful than the implicit actions, they fulfil the general purpose requirement).

Y.P. Yung [Yun93, p.92] follows this distinction, repeating the example of display-updating actions in a spreadsheet and again naming this 'implicit' action:

> We can imagine that there are implicit actions which update the values of the variables on the screen. Similarly, there are such implicit actions in the implementations of definitive notations. For example, each graphical object in DoNaLD has a representation on the screen. For a `point` variable there is a dot to represent it; for a `line` variable, there is a linear set of points and so on. So there will be a `plot_point` action in the implementation of DoNaLD to be called automatically when a `point` variable is updated, and likewise a `plot_line` action for a `line` variable.

However, the terms 'implicit' and 'explicit' have been used in different senses since. The implicit/explicit distinction in *action* is in fact the reverse of a distinction sometimes made between implicit and explicit *dependency*. This distinction is commonly made in sources on object-oriented programming. Cartwright, in the overview documentation for his JaM2 API (*jam2Cartwright2001/JaM2/overview.html*) which

follows on from his thesis work [Car99], defines the terminology in the following way:

> Two distinct forms of dependency exist, explicit and implicit:
>
> - explicit — Dependency is expressed in a well-defined underlying algebra between values of simple data types such as numbers and strings of characters, using well-defined functions over those data types. An actual spreadsheet is an example of explicit dependencies (although the way they are maintained internally by the spreadsheet executable may not be).
> - implicit — Dependency is expressed through a system of messages sent to software objects (class instances or a pointer to a value of an abstract data type) to signal when they should update. This process relies on code inside procedures or methods performing the correct update. The `make` application that is used to maintain dependency between source files for an executable when it needs to be rebuilt is an example of this.

Heron [Her02, pp.16–17] clarifies this further:

> Explicit dependency is that found within a spreadsheet and within definitive scripts, it relies on a formal notation. The spreadsheet definitive machine can interpret and work out the dependencies from the script itself. Implicit dependency as found in a `Makefile` relies on side effects caused by procedural commands (i.e. compiling). Since there is no formal notation to describe these side effects then the dependency maintainer (in this case `make`) requires that the dependencies are spelt out to it.
>
> Traditional programming makes no use of explicit dependency and any dependencies that exist (whether they are message passing, propagating updates, etc) require specific procedural actions to be written and maintained by the programmer to keep the dependencies valid.
>
> ... The most successful and widely used Empirical Modelling tools have combined the powerful explicit dependency of definitive scripts with the less formally defined implicit dependency of procedural programming. Tools such as EDEN combine definitive notations with functions and procedures in a similar way to the modern programmable spreadsheet.

In Eden, therefore, explicit dependency is maintained by implicit actions. Explicit actions may be used to maintain implicit dependency.

Authors therefore vary in their meaning of the terms implicit and explicit. Sources (including, but not limited to the above) variously emphasise the following:

- Formality of notation (hence, analysability) used to describe when the action should occur;
- Instigator of the action: the system or the user?
- Specifier of the action: the tool implementer or the tool user?

- The 'defined-ness' of the references used: are literal values stated or references to other values?

In the sections below, the words implicit and explicit are (explicitly!) avoided. It is necessary however to build on the terms initially defined in Appendix §3.A (p.178). The sections below extend the use of the terms below from the script graph context to Eden definitions and actions in the following way:

Triggers of change are what *cause* change;

Targets of change are what the change will *affect*;

Sources are what an action *observes*.

The terms sources and triggers are synonyms where definitions are concerned, since definitions observe only their triggers, as will become clear.

4.3.2 Requirements of an EDEN-like definition maintainer

With the benefit of examining the ADM and the DAM machine, I can now state requirements for an EDEN-like definition maintainer (DM) that are firmer than Y.W. Yung's rule of thumb.

When redefinition(s) are made, a definition maintainer must re-evaluate the necessary atoms and arrive at a final steady state. In general, the redefinitions can be a *set* of changes. The BRA is explicitly designed to process a set of redefinitions. EDEN has an '`autocalc`' variable that can be used to form a set of redefinitions.

In Figure 4.32, I state three types of requirements for an EDEN-like DM. The requirements are necessary for correctness, necessary for efficiency or may be desirable respectively.

Necessary requirements

1. In the final stable state, (at least) all target atoms of the change set must have been re-evaluated.
2. In the final stable state, each atom must have been re-evaluated *after* its source atoms.

Requirements for efficient performance

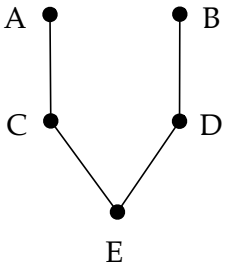
3. Only atoms that are targets of the change set need be evaluated. (This is a corollary of requirement 1.)
4. An atom should not be re-evaluated more than once during a single transition.

Possibly desirable features

5. We may want to separate definitions and actions, evaluating definitions before the actions so that the actions observe consistent definitive state. Alternatively this can be thought of as the definitions taking priority over the actions. (Y.W. Yung suggests this as a way of separating the 'mathematical' and 'feedback' parts of a model.)
6. We may wish to minimise the total memory used. A recursive solution requires more stack memory as it uses a nested calling mechanism.
7. We may wish to minimise the amount of computation performed. A non-recursive solution will have a time-consuming sorting phase.
8. We may want to detect and prevent cyclic dependency.

(Note: the 'sources' and 'targets' referred to above are determined from the script graph *after* the set of redefinitions have taken effect.)

Figure 4.32: Requirements of an EDEN-like definition maintainer



Depth-first, left to right: ECADB
Depth-first, right to left: EDBCA
Breadth-first, left to right: ECDAB
Breadth-first, right to left: EDCBA

Figure 4.33: Depth- and breadth- first topological sorts

Requirements 2 and 4 in Figure 4.32 can be achieved by a topological sort, which is an operation that embeds the partial ordering described by a script graph (see Appendix §3.A, p.178) into a linear ordering which can then be used as an evaluation sequence. Two ways of implementing a topological sort (*cf.* §3.1.2) of the nodes of a script graph are depth-first and breadth-first, which are usually implemented recursively and iteratively respectively. To fully determine the ordering, it is also necessary to know the order in which the edges emanating from each node are processed. A basis for this ordering could be established on a geometric basis with reference to a figure (e.g. leftmost to rightmost) or on a temporal basis (e.g. oldest to newest). Figure 4.33 shows four different topological sorts for a particular graph.

4.3.3 Black-box analysis of EDEN

This section gives an analysis of the current EDEN DM by systematically constructing various definitive scripts and observing how they are re-evaluated. The approach taken here could be used to evaluate any other DM.

Firstly, scripts containing only definitions are investigated. Definitions in Eden can use any user-defined function. It is possible to write a user-defined function which includes side-effect(s). This means it is technically no longer a function, but if we only print literally defined state, this will not affect the DM scheduling.

The scripts shown in Figure 4.34 therefore use a user-defined function to print a trace when evaluation occurs, making evaluation visible. The results are explained below. They have been obtained by running the scripts in `ttyeden-1.46`.

Script 1 is a fully constrained evaluation ordering. When '`d`' is changed (denoted in the diagram by the bold font), the only possible correct way to evaluate script 1 is '`c`', '`b`', '`a`'. EDEN evaluates this script in this way.

Script 2A in Figure 4.34 reveals a choice of breadth- or depth-first evaluation orderings, as illustrated in Figure 4.33. EDEN evaluates this script using a breadth-first ordering. Script 2B shows that EDEN is using a temporal basis for edge ordering: when there is a set of atoms with no ordering constraint mandated by the script graph, EDEN evaluates them in the order of their (re)definition, oldest to newest.

Script 3 could lead to double evaluation in some DM implementations. EDEN however evaluates the definition '`a`' last, complying with the necessary requirement (2) from Figure 4.32, and evaluates it only once, complying with the efficiency requirement (4).

Scripts 4A, 4B and 4C show the results of making changes to a *set* of definitions. In each example, the multiple changes could lead to multiple unnecessary evaluations. In Eden, a set (or, in Cartwright's terms, a block) of redefinitions is formed using the `autocalc` variable. Script 4C (which also appears as the 'power' example in Figure 3.2, p.115 *et seq.*) is taken from Cartwright's thesis. Contrary to Cartwright's assertions about `tkeden` (quoted in §3.1.2) that EDEN "...often performs a large number of unnecessary calculations", EDEN in fact optimises re-evaluation across the set of changes. This result shows that the BRA has no particular advantage over the algorithms used in EDEN (other than that the BRA is well documented by Cartwright — itself a big advantage). This finding reinforces the conclusions reached on the basis of performance measurements comparing !Donald and EDEN in §3.4.4.

Continuing systematically, we now investigate scripts containing only actions, in order to examine the evaluation strategies employed by EDEN here. We shall reuse the examples just presented, translating the definitions into actions.

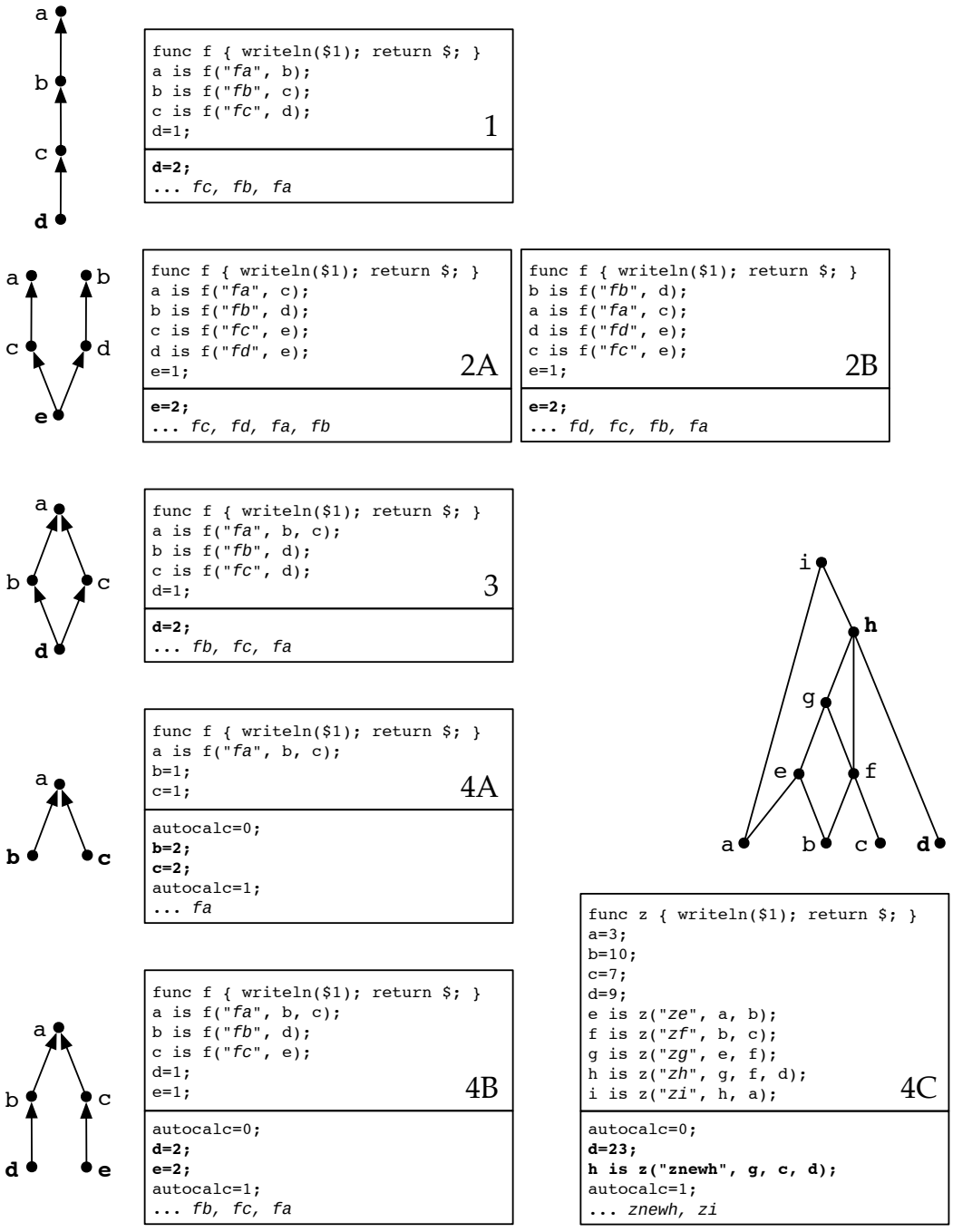


Figure 4.34: Definitive script graphs containing only definitions systematically constructed to examine DM evaluation strategies

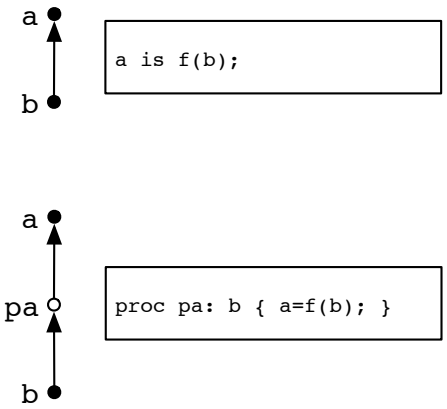


Figure 4.35: An equivalent definition and action

In Eden, the value of an action is the action declaration itself. The actions therefore cannot represent the values of the definitions directly. Translation from definitions to actions therefore involves the introduction of more atoms: the actions themselves. The introduction of the extra atoms is the topic of §5.1.1. Here, we make the following transformation. The definition:

```
a is f(b);
```

can be transformed to the action:

```
proc pa: b { a = f(b); }
```

The diagrammatic conventions used here for this transformation are shown in Figure 4.35 (which, like Figure 4.34, does not show the function `f`).

The examples in Figure 4.36 show that when scripts containing only definitions are translated to only actions using the transformation scheme shown above, then the actions are evaluated in the same way as the original definitions were. This is what Y.W. Yung means by the 'equivalence' of definitions and actions.

It may seem strange that EDEN manages to order the evaluation of the actions as well as the definitions, despite the DM lacking knowledge about what actions actually write to. We shall see that the EDEN scheduler actually schedules definitions and actions in the same way, and in a way that does not require full knowledge of the consequences of an action.

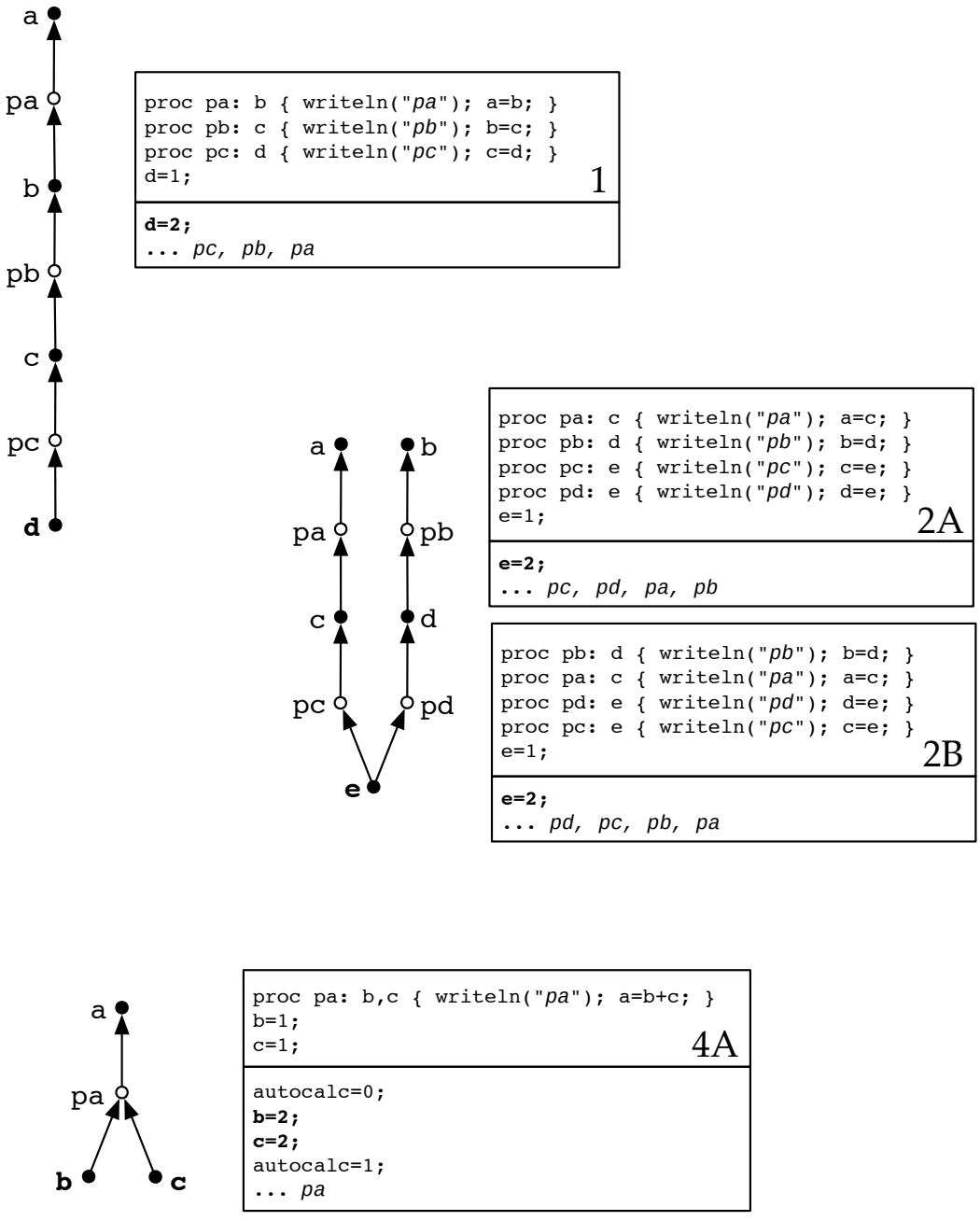


Figure 4.36: Definitive script graphs containing only actions systematically transformed from Figure 4.34, which contains only definitions

4.3.4 Differences between definitions and actions in Eden

Having examined the 'equivalence' of definitions and actions under the described transformation scheme, we are now in a position to elaborate the differences between definitions and actions more clearly. The limits of the transformation scheme used show that actions can do more than definitions. This gives us insights about *both* definitions and actions.

Constraints on definitions in Eden

A. A definition writes only to a single well defined atom, stated on the LHS of the '`is`' (the target).

B. A definition is the only agent in the system that writes to the atom specified on the LHS of the '`is`' (the target).

C. A definition reads only from atoms stated on the RHS of the '`is`' (the sources/triggers).

Freedom of actions in Eden

D. An action can write to any atom, including something stated on the RHS of the colon (the triggers). (Compare (A).)

E. An action can write to any *set* of atoms. (Compare (A).)

F. It is not possible to tell what an action writes to without executing it. (Actually, it might be possible in some limited cases, but the current EDEN makes no attempt to tell.) (Compare (A).)

G. It is possible for a single atom to be written to by more than one action. (Compare (B).)

H. An action can read from an atom that is not stated on the RHS of the colon (a trigger). (Compare (C).)

4.3.5 Inside the machine: definitions and actions

Looking at the source code, the core of the EDEN machine is actually a three line segment of C code, which was a part of the original `hoc` implementation (by Kernighan and Pike — see §4.1.1). It is shown below, slightly simplified.

```
while (*pc) {
  (*(*pc++)) ();
}
```

In informal English, this reads:

> Whilst the function pointer (or virtual opcode) at location `pc` is not 0:
>
> Call the function pointed to (involving a double dereference),
>
> Increment `pc` to point to the next function pointer in the array.

This core is a virtual machine which executes an instruction stream of function pointers, rather as the Java Virtual Machine executes bytecodes.

Definitions and actions are both encoded into streams of instructions which are executed in order to enact state change when the scheduler deems it necessary. The execution mechanism is the same for both definitions and actions. We can therefore think of definitions as being maintained by *definition agents*.

What then, is the technical distinction between definition and action? There are two parts to the answer. The first part has been given in the previous section. Points A to C represent *constraints on definition agent action*. The second part of the answer involves priority in scheduling. Roughly speaking, definitions take priority over actions so that definitive state always appears consistent. The full detailed picture is given in the next section.

Restrictions on reading and writing to state based on agent identity have long been at the centre of abstract data type and object-oriented thinking. Definitive systems however use the script graph as a basis for restrictions rather than modules, and the restrictions are combined with evaluation scheduling and prohibition of graph cycles.

4.3.6 The EDEN scheduler algorithm

Although largely explained without reference to the source code, the insights in the previous sections have been derived with the help of detailed inspection of the code (which as Figure 4.9 on p.219 shows, is currently 30,000 lines of C).

I have reduced the source code representing the scheduler algorithm (which is spread across five source files in the present code) down to the pseudo-code shown

in Appendix 4.A on p.277. I have renamed some of the functions for clarity — the renaming is documented in the box numbered 5.

From the description of the algorithm, I have produced the diagram showing the EDEN machine data, operations, data flow and control flow shown in Figure 4.37. This diagram reveals part of the difficulty of analysing the code: many operations lead to the `execute` procedure at the base of the figure, corresponding to the three line virtual machine implementation described in the previous section. The virtual machine can execute any function pointer, and here one of the difficulties arises — many of the function pointers correspond to a re-entrant call into the machine, shown as a line looping from the bottom to the top of the figure. Still, the diagram is helpful for much analysis about the EDEN scheduler.

The final reduction I have made to describe the EDEN scheduler is shown in Figure 4.38. This set of diagrams is an attempt to represent the operational contexts of the EDEN machine more statically than in Figure 4.37. It clearly shows the dual nature of the EDEN machine, which evaluates both speculatively[25] and on demand. Demand-driven evaluation occurs when the statement being executed reads from definitive state that is marked as OutOfDate. Speculative evaluation is initiated by change, and can be inhibited by changing the setting of `autocalc`. This is how sets of redefinitions are processed efficiently: when speculative evaluation is inhibited, changes still cause scheduling of necessary definitions and actions, but the evaluation (diagram 3 in Figure 4.38) is not actually performed.

Another major insight contained in these representations of the EDEN scheduler is that EDEN can be viewed as a stack of virtual machines, each machine implementing an indivisible view of state for the machine above. There are presently three such virtual machines: definition processing is at the bottom, actions execute over definitions, and an interface automation layer executes over actions. Thus, a triggered action observing definitive state will always observe it in the UpToDate state[26]. We have seen that the scheduling of definitions and actions is very similar — but the scheduling operates on two different levels. One layer up, the interface will only display states that occur after processing of actions has completed — stated

[25] On the assumption that an UpToDate variable will be observed before it is changed again.
[26] Although there is a caveat here to do with adjacentSourcesAreUpToDate of the definitive state.

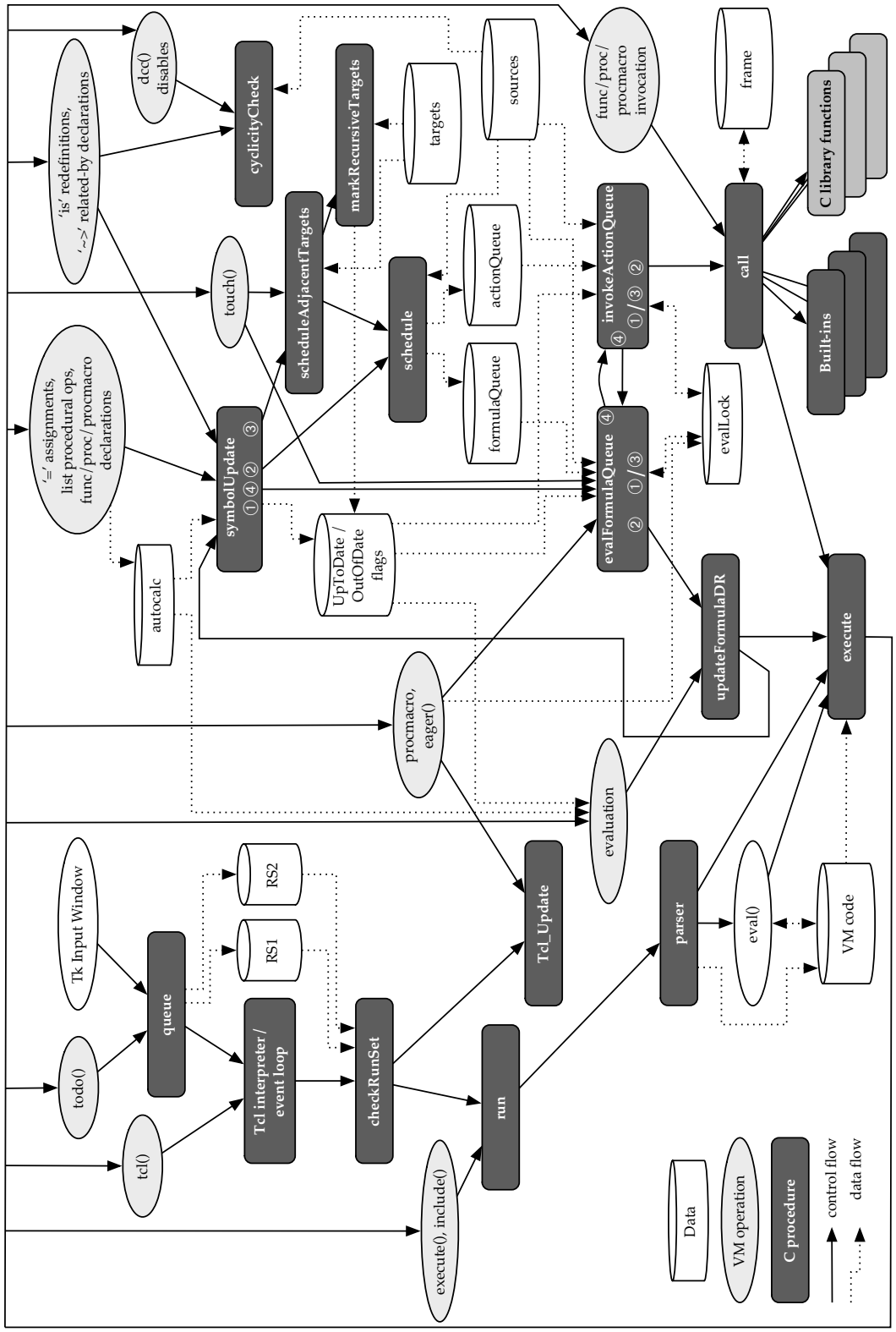


Figure 4.37: EDEN machine data, operations, data flow and control flow

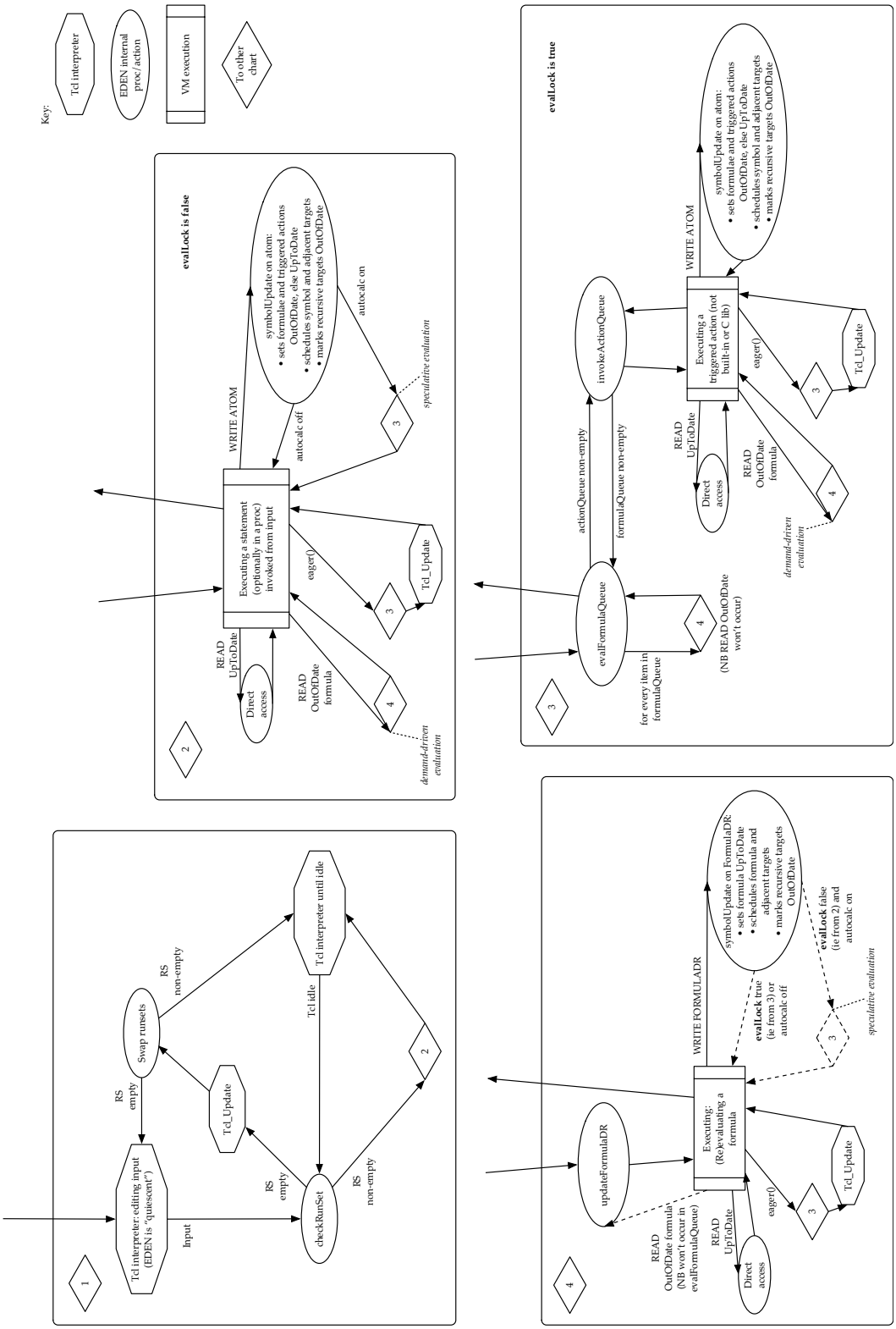


Figure 4.38: The EDEN machine states considered more statically

conversely, the interface will never show the state mid-action[27]. This structure is similar to that suggested by Dijkstra in his "Notes on Structured Programming", where he expresses his wish for meaningful state when a virtual machine is between instructions. He states [DDH72, pp.49–50]:

> ... we have arranged our program in layers. Each program layer is to be understood all by itself, under the assumption of a suitable machine to execute it, while the function of each layer is to simulate the machine that is assumed to be available on the level immediately above it.
>
> ...
>
> We may hope that the model will give us a better grip on the problems that arise when a program has to be modified while it is in action. If a machine at a given level is stopped between two of its instructions, all lower machines are completely passive and can be replaced, while all higher machines must be regarded as engaged in the middle of an instruction: their state must be considered as being in transition. In a [conventional] sequential machine the state can only be interpreted in between instruction executions and the picture of this hierarchy of machines, each having its own instruction counter — "counting its instructions" — seems more profitable if we wish to decide at any given moment, what interpretations are valid. In the usual programming language in which computational progress is measured in a homogeneous measure — say "the grain" of one statement — I feel somewhat helpless when faced with the question of which interpretations are valid when.

My current best conceptual model of EDEN execution follows this notion of layers of indivisibility. In this model, an action making a change to a variable value can cause the definition layer to gain priority and bring the definitive state back into consistency, followed by resumption of the previous action at the previous level. A transition of priority in the reverse direction can also occur if a change is propagated via dependency to a value which is a trigger for an action. Transitions of priority in this direction however do not take place immediately — actions are queued until the definition layer has finished executing. This conceptual model is a helpful guide (and it also makes the sequential nature of EDEN clear), but it is unlikely to capture all of the actual behaviour of the current scheduling algorithm.

Unfortunately the actual layers (such as they are) are only made visible in the current implementation when diagrams such as Figure 4.38 are drawn. The notion of layers may be helpful in planning a new implementation, however. Y.P. Yung in fact makes such a proposal in [Yun96] which will be examined in the next section.

[27] There is a caveat here too, as the `eager()` command can be used to force the current state to be propagated to the interface.

4.3.7 Higher Order Definitions (HODs) in EDEN

This section addresses the question: does EDEN need another layer in order to cope with HODs?

Y.P. Yung proposes [Yun96, p.27] but does not fully explain a "4-layer prioritised action system" in an attempt to rationalise the multiple action control sub-systems in EDEN. The proposal comprises a 'definition layer', 'meta-definition layer', 'decision layer' and 'transition layer'. The definition layer maintains the state specified by definitions. In this layer, multiple triggers may be safely eliminated. The meta-definition layer maintains Higher Order Definitions[28] where "because the aim is still to maintain the consistence of state, hopefully we can eliminate multiple triggers". The decision layer is needed "to determine what actions are needed to transit the state... Actions in the Decision layer may still interfere. However it becomes apparent that this kind of interference is the same class as the interferences between concurrent agents that we are more comfortable to live with." Finally, the transition layer actually performs the actions determined by the decision layer.

I did initially believe that a HOD layer was necessary, but I thought that it would need to be added at the bottom, since HODs change the script graph, which is assumed to be static by the definition layer. After having analysed the EDEN scheduler at the level shown in the previous section, however, I no longer believe such a layer to be necessary. The key realisation here is that the EDEN scheduler only schedules adjacent targets of any change (although it marks *all* targets recursively as OutOfDate). As it does not construct a deep schedule, it is flexible enough to cope with actions changing the script graph.

A standard example in this thesis is the '`if`' HOD:

```
v is b ? x : y;
```

(in English, if `b` is true, `v` is `x`, otherwise `v` is `y`). The problem with the definition as stated is that `v` is dependent upon both `x` and `y`, whereas the dependency should be upon *either* `x` or `y`, the choice itself dependent upon the current value of `v`. The following example illustrates the problem.

[28] HOD is the term used by [Yun90, p.103] to describe "a definition of a set of definitions". Y.P. Yung [Yun93, p.115] calls these "meta-definitions (structures that generate definitions)", but the term HOD has continued in usage since.

```
v is b ? x : y;
b is 1; x is 2; y is 3;
proc p : v { writeln("v has changed"); }
x is 4; /* p is triggered */
y is 5; /* ! p is triggered, incorrectly */
```

With the added confidence in EDEN's scheduler, however, we can write the following:

```
proc cv: b { if (b) { v is x; } else { v is y; } }
```

This action reads very much as the English description. It redefines `v` when `b` is changed. Such Eden code has not seemed popular in the past — and there were in fact some bugs, initially discovered by Carlos Fischer (a visiting researcher), relating to triggered actions that make redefinitions, which I fixed in version 1.13. Perhaps we can now start writing HODs in this style.

4.A Pseudo-code of the EDEN scheduler algorithm

```
RUNSET ------------------                                    1

User input from Tk Input Window,
todo() input
call...

  queue(cmd)
    add cmd to the tail of the RS list
    Tcl_DoWhenIdle(checkRunSet)

  checkRunSet()
    if (interrupted)
      clear run sets

    if (RS is not empty)
      while (RS is not empty)
        get an action string, s, from the head of the RS list
        run(s)
      Tcl_DoWhenIdle(checkRunSet)

    else
      Tcl_Update
      swap runsets
      if (RS is non-empty)
        Tcl_DoWhenIdle(checkRunSet)


PARSER ------------------

execute(), include() and many other routines that need to invoke the
parser on a string or a file call...

  run(s)
    Invoke the parser on string s to generate VM code p
    execute(p)
```

```
CYCLICITY CHECK ---------                                    2

'is' redefinition,
'->' related-by declaration
call...

  cyclicityCheck(sp, sources)
    Recursively search the sources list for an occurrence of sp.  Mark
      the sources encountered along the way in order to determine when
      the job is done.  When the job is done, search again and remove
      the markers.  The Eden function dcc() can be used to globally
      disable these checks.


TRIGGER -------------------

'=', 'l=l//q', '+=', '-=', assignment,
'++', '--' post/pre increment/decrement,
'shift', 'append', 'insert', 'delete' list procedural ops,
'func/proc/procmacro' declaration,
'func/proc/procmacro' triggered action declaration,
'is' redefinition,
'->' related-by declaration,
... all VMOPs
call...

  symbolUpdate(sp)
    if (sp is a newly redefined triggered action or definition, defined
        by the caller)
      mark sp OutOfDate
    else
      mark sp UpToDate
    schedule(sp)
    scheduleAdjacentTargets(sp)
    if (autocalc)
      evalFormulaQueue


procmacros,
use of Eden eager() procedure
call...

  eager() VMOP
    save the value of evalLock
    evalLock = FALSE
    evalFormulaQueue
    restore the value of evalLock
    Tcl_Update


  touch()
    for each argument, sp, provided to touch()
      scheduleAdjacentTargets(sp)
    evalFormulaQueue
    pushUNDEF
```

```
SCHEDULE ------------------                                    3

  schedule(sp)
    if (sp is a formula)
      Q = &formulaQueue
    else if (sp is a proc, func, procmacro or built-in)
      Q = &actionQueue
    else
      return

    if (sp is already in Q)
      move sp to the end of Q
    else
      if (sp is a formula)
        append sp to the end of Q
      else if (sp is a proc, func, procmacro or built-in)
        if (sp has sources) -- ie it is an action
          append sp to the end of Q


  scheduleAdjacentTargets(sp)
    for each adjacent target, t of sp
      markRecursiveTargets(t)
        schedule(t)


  markRecursiveTargets(sp)
    if (sp is UpToDate)
      mark sp as OutOfDate
      for each adjacent target, t of sp
        markRecursiveTargets(t)


EVALUATE ------------

  evalFormulaQueue()
    if (evalLock is TRUE)
      return

    evalLock = TRUE
    while (formulaQueue is not empty)
      remove the front item, sp from formulaQueue
      if (sp is OutOfDate)
        if (adjacentSourcesAreUpToDate(sp))
          updateFormulaDR(sp) // this may add items to the formula and
                              // action queues
    evalLock = FALSE
    // the formulaQueue is now empty

    if (the actionQueue is not empty)
      invokeActionQueue
    // the actionQueue and formulaQueue are now both empty
```

```
  invokeActionQueue()                                          4
    if (evalLock is TRUE)
      return

    evalLock = TRUE
    while (actionQueue is not empty)
      remove the front item, sp from actionQueue
      if (adjacentSourcesAreUpToDate(sp))
      call(sp, no arguments) // this may add items to the formula and
                             // action queues
        discard the value returned
    evalLock = FALSE
    // the actionQueue is now empty

    if (the formulaQueue is not empty)
      evalFormulaQueue


  lvalue evaluation, l=l//q optimisation, back-ticks
  check for an OutOfDate formula and sometimes autocalc
  and call...

    updateFormulaDR(sp)
      if (UPDATE) -- used to prevent evaluation by the ? query operator
        execute(VM code associated with sp)
        store the value returned in the data register associated with sp
        symbolUpdate(sp)


  func/proc/procmacro invocation
  call...

    call(sp, arguments)
      save information about the current frame

      if (sp is a proc, func or procmacro)
        push the arguments
        push UNDEFs for local variables
        execute(VM code associated with sp)
      else if (sp is a built-in or c library function)
        push the arguments
        appropriately invoke code associated with sp

      restore frame information
      leave the item returned on the stack


  eval(expr) calls execute on expr before storing the result,
  back-ticks
  call...

    execute(p)
      execute VM function pointer code at p,
        until a rts (zero) VM code is found

        ... NB VM code can include any VMOPs
```

5

```
NOTES
  · Some procedure arguments have been simplified
  · sp is OutOfDate is when sp->changed=TRUE,
      UpToDate when sp->changed=FALSE
  · This pseudo code omits mention of the "master" stack of agent names
  · setprompt() has been omitted from checkRunSet()
  · Tcl_Update is actually Tcl_EvalEC(interp, "update")
  · symbolUpdate simplifies setting of OutOfDate flag by caller

  HERE NAMED                ACTUAL NAME          FILE LOCATION

  formulaQueue              formula_queue
  actionQueue               action_queue
  evalLock                  lock
                            queue                [main.c]
                            checkRunSet          [main.c]
  cyclicityCheck            checkok              [refer.c]
  symbolUpdate              change               [eval.c]
  spValueFromISRef          addr                 [machine.c]
  spValueFromStackRef       getvalue             [machine.c]
  spValueFromStackStringRef lookup_address       [machine.c]
                            eager                [eval.c]
                            touch                [builtin.c]
                            schedule             [eval.c]
  scheduleAdjacentTargets   schedule_parents_of  [eval.c]
  markRecursiveTargets      mark_changed         [eval.c]
  evalFormulaQueue          eval_formula_queue   [eval.c]
  invokeActionQueue         invoke_action_queue  [eval.c]
  updateFormulaDR           update               [machine.c]
                            call                 [code.c]
                            execute              [code.c]
  adjacentSourcesAreUpToDate ready               [refer.c]
```