Chapter 5

Problematic issues in dependency maintenance

This chapter pulls together three deep problems in dependency maintenance that run through the earlier chapters in this thesis, which previously had the status of ill-understood research problems. I make some specific proposals which help to deal conceptually with each problem. The proposals do not constitute detailed designs, but they do transform ill-defined abstract research problems into precise technical problems requiring an engineering solution that will be the subject of future tool development.

The three topics are those of concurrent definition maintenance, moding, and Higher-Order Definitions (HODs). The topics are somewhat intertwined. Each topic is treated below separately, but the discussion leads on from one to the next.

5.1 Concurrent definition maintenance

The design of a concurrent definition maintainer involves two principal issues. These issues are considered separately in the subsections that follow below.

1. Determining how to map evaluation agency to a set of 'definition-agents'.

2. Specifying how the concurrent evaluation should be synchronised.

The job of a definition maintainer involves scheduling evaluations of definitions. Exactly when evaluations are scheduled depends partly on the choice of evaluation/storage strategy — evaluations can be performed on use, redefinition, or some mix of the two. (This was described in §2.2.1 and `am`, the DAM machine and EDEN are examples of each particular case.)

The specifics of a particular evaluation — the scheduling order of minor transitions within a major transition — depends on two things:

A. the structure of the script digraph, as described by the corresponding level assignment[1], and

B. which particular nodes are being redefined[2].

Within a major transition, there is sometimes potential for certain minor transitions (see §2.3.3) to occur concurrently. Specifically, within the set of nodes that require re-evaluation, all nodes that have the same level assignment may be evaluated concurrently. The potential for concurrent evaluation varies with the script. For example, N5A4ag (in Figure 3.28, see p.189) has a completely constrained evaluation ordering with no potential for concurrency. In contrast, after a change to the leaf in N5A4a (p.189), all the root nodes can be evaluated concurrently.

5.1.1 Mapping evaluation agency to definition-agents

Firstly, let us just consider issue 1. How many concurrent processes do we require and what part of the evaluation does each process perform?

Eden is a language that can describe both dependency and agency. One possible transformation of a definition to an "equivalent" evaluating agent was described using the Eden language in §4.3.3. When a set of definitions is considered, there are many possible ways that the transformation can be made. A good way to describe this is to extend the script digraph to take account of the evaluating agency of the definition maintainer. I describe this as adding *definition-agents* to the script digraph.

[1]See Appendix §3.A, p.178 for Harary's definition of 'level assignment' and §3.1.2 for Cartwright's BRA — an algorithm based on Knuth's topological sort algorithm that calculates an evaluation ordering consistent with that described by a level assignment.

[2]Section §4.3.2 gives a description of which nodes require re-evaluation after a change.

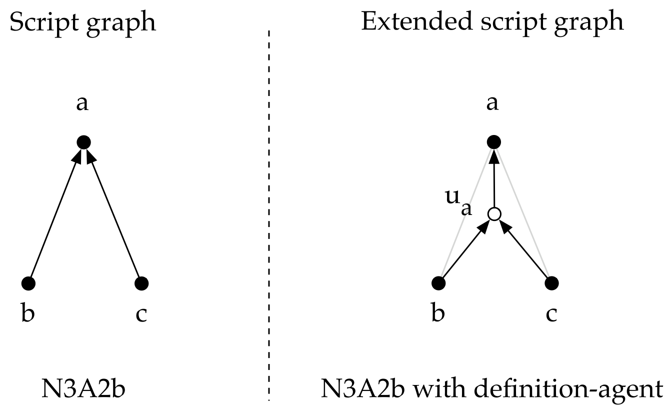


Figure 5.1: The script graph for `a is b+c`, together with an associated extended script graph devised by adding a definition-agent node

The script digraph for the one-line definitive script `a is b+c` is shown on the left of Figure 5.1 (reproduced from N3A2b in Figure 3.28 on p.187). Taking a 'dependency-as-agency' perspective (i.e. considering the way in which dependency is implemented through agency), we can transform the definition into Eden's alternative[3] triggered action form[4]:

```
proc ua: b, c { a = b+c; }
```

Now the agency involved in updating the variable '`a`' can be represented by adding a new type of node, to be called a *definition-agent node*, to the script digraph diagram. The extended script graph diagram, shown on the right of Figure 5.1, reflects the dependency-as-agency perspective.

An extended script graph is a *bipartite*[5] digraph. It has two types of node: the original *value nodes* and the new definition-agent nodes. It also has two types of arc. Arcs whose starting location is a value node (such as the arc $\mathtt{b} - \mathtt{u_a}$) represent a *read* operation by the definition-agent. Arcs whose starting location is an definition-agent (such as the arc $\mathtt{u_a} - \mathtt{a}$) represent a *write* operation by the definition-agent.

[3]But not completely equivalent — see §4.3.4.

[4]I use a subscript — which is not possible in the real Eden — to make clear which variable the action is updating.

[5]A graph G is bipartite if the nodes of the graph can be split into disjoint sets A and B such that each edge of G joins a node of A and node of B [Wil96, p.18].

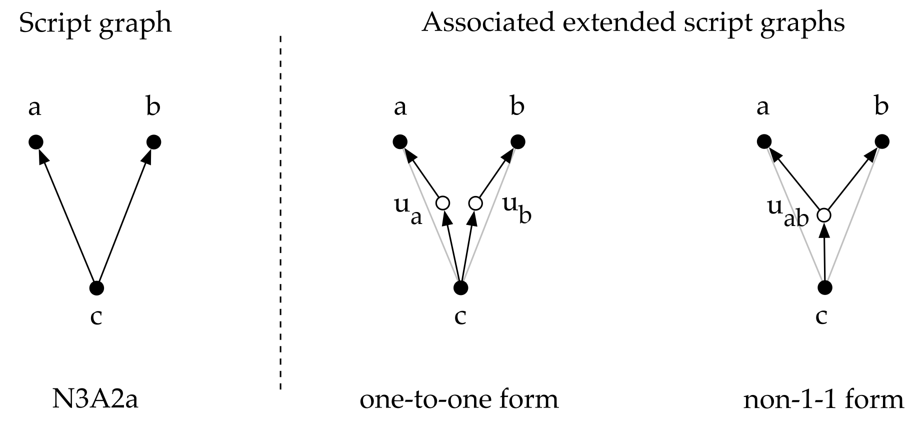


Figure 5.2: Two possible definition-agent arrangements for a simple two-line definitive script

In most non-trivial cases, there is more than one possibility for the total number and locations of definition-agents. Consider the script digraph shown on the left of Figure 5.2 (reproduced from N3A2a in Figure 3.28 on p.187), which represents the dependency structure in the two-line definitive script:

```
a is c;
b is c;
```

It is possible to maintain these two dependencies with either one or two definition-agents. Therefore, there are two possible triggered action forms of this definitive script: a *one-to-one* form where each definition is mapped to a definition-agent:

proc u_a: c { a = c; }

proc u_b: c { b = c; }

or a *non-1-1 form*, where more than one definition may be mapped to a single definition-agent:

proc u_{ab}: c { a = c;[6] b = c; }

There are two corresponding extended script graph diagrams for these two scripts, shown on the right of Figure 5.2.

[6]These two operations need not be performed sequentially, but there is no way to specify this in the present Eden.

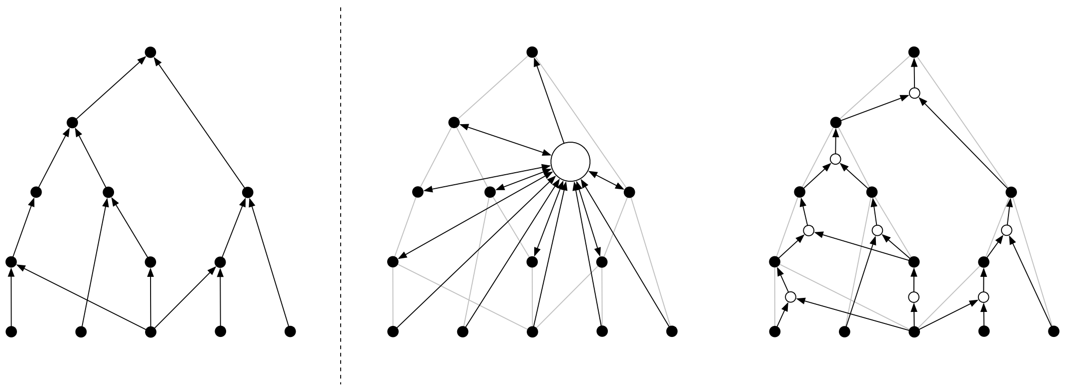


Figure 5.3: A script graph, the corresponding "monolithic" and one-to-one configurations of definition-agent nodes

Most definition maintainers can be considered to have only one definition-agent which is a "monolith". In this type of analysis, the definition maintainer can be considered to be an agent of the form:

```
proc u_all: all { schedule updates; do updates }
```

Figure 5.3 shows the distinction in diagrammatic form. A script graph is shown on the left. The middle of the figure shows the configuration of the single monolithic definition-agent in most definition maintainers. The monolithic definition-agent reads from leaves; writes to roots; and both reads from and writes to inner nodes. The DAM machine is an example of a DM with a monolithic definition-agent. In contrast, on the right, a one-to-one configuration of definition-agent nodes to non-leaf nodes is shown.

Each extended script graph describes a possible way of allocating update evaluations to agents. Figure 5.3 illustrates allocations at two extremes, where all updating is assigned to a single agent or where each update is handled by a separate agent. Many mappings between these two extremes are possible. The possible mappings of nodes of the script graph to definition agents provide us with a conceptual means with which to think about the number of concurrent processes required and what part of the evaluation each process should perform.

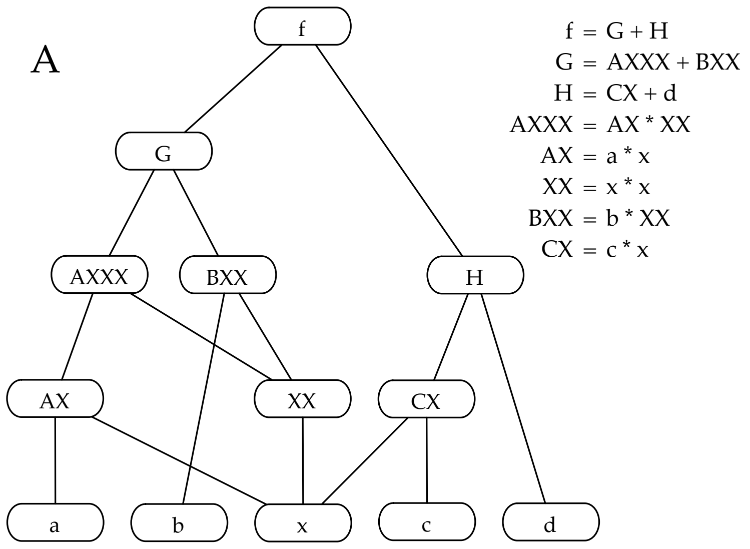
A previous reference that discusses the mapping of evaluation agency to concurrent processes is [Yun90, §7], where Y.W. Yung suggests (although not in these terms) that a definition-agent matches the abstract view of a node in a multicomputer system, consisting of multiple nodes connected by a network, each comprising a CPU and some memory.

> A definition in definitive languages can be viewed as a composition of *data* and *program*:
>
> ```
> definition = data + program
> ```
>
> where *program* is the method of evaluating the *data* and is expressed in mathematical terms. The *program*, theoretically, has no interference with other definitions since the only thing affected is the value (*data*) of the *definition*. This perspective on a definition matches the abstract view of the node in the multicomputer system:
>
> ```
> node = memory + CPU
> ```
>
> where *memory* stores *data* and *CPU* executes *program*. The data dependency of definitions describes the links of the nodes.

The dependency-as-agency discussion above has allowed us to go further than this "definition as node" perspective to show the wide variety of possible mappings of evaluation onto agents in concurrent dependency maintenance.

Y.W. Yung takes the discussion further in a different direction, pointing out that a single definition can be decomposed into sets of simpler definitions, which may create further potential for concurrent evaluation. In the terms used here, this corresponds to decomposing a single node in the script graph into a sub-graph of components. Figure 5.4 shows Y.W. Yung's example of two different decompositions of the definition $f = ax^3 + bx^2 + cx + d$ (*cf.* Figure 4.16 on p.234, which illustrates different decompositions of output from the EDDI AOP).

On what basis are these various mapping decisions to be made? It seems that (similar to the basis for the decision of evaluation/storage strategy) the nature of evaluation and change in the application are important here. Although the script graph labelled B in Figure 5.4 has a larger total of level assignments and hence appears to have a smaller potential for concurrent evaluation, Y.W. Yung points out that in the special case where `a`, `b`, `c` and `x` are seldom changed but the value of `d` varies frequently, script B is likely to involve fewer evaluations than script A.



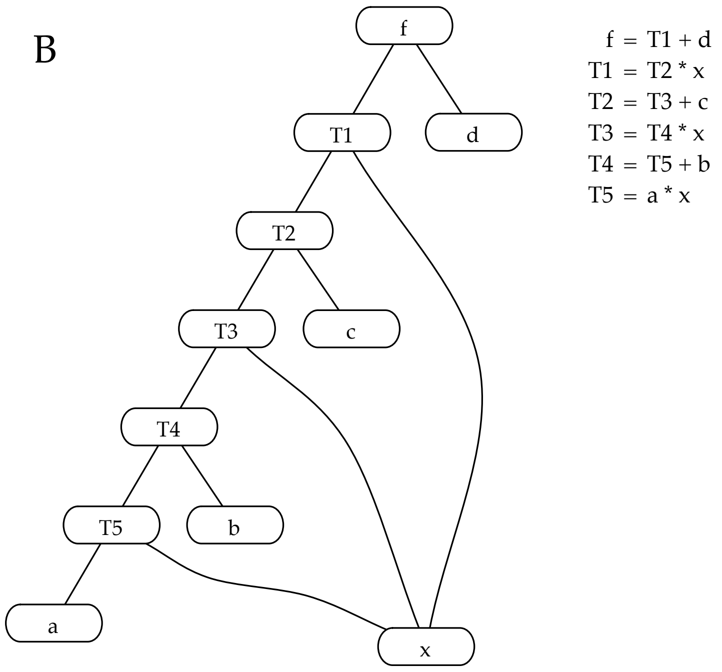


Figure 5.4: Decomposition of the polynomial definition $f = ax^3 + bx^2 + cx + d$ into two possible definitive scripts (from [Yun90, pp.97–98])

5.1.2 Synchronisation of concurrent definition-agents

The previous section has shown how evaluation can be segmented into concurrent parts. This section attempts to address the necessary synchronisation between those parts.

If we are motivated to use concurrency to speed up an existing sequential definition maintainer, then we do not need to examine the concept of dependency too closely — in this view, definitions describe potential parallelism in update. The necessary synchronisation is determined by the level assignments in the script graph. Within a major transition, necessary evaluations must be ordered by level assignment in order to produce values that are consistent with their definition. Within the set of nodes that require re-evaluation, all nodes that have the same level assignment may be evaluated concurrently.

A version of Cartwright's JaM2 Java API implements a form of *concurrent update* along these lines. In JaM2, like the DAM machine, redefinitions are queued until an '`update`' routine is invoked. The `update` routine locks access to the store of maintained state, checks the set of redefinitions for graph cycles, forms a schedule and then proceeds with evaluation, evaluating multiple definitions simultaneously in multiple threads where possible. When evaluation is complete, the store is then unlocked [Car04].

Y.W. Yung [Yun90, §5.3] also briefly describes a scheme for a concurrent definition maintainer employing message passing, nodes sending MARK, ACKNOWLEDGE, EVALUATE, QUERY and ANSWER messages to other concurrently operating nodes in order to propagate change.

Neither of these schemes considers the indivisible nature of dependency too closely, however. Beynon, Cartwright, Sun and Ward [BCSW99] contains the following characterisation of dependency, which is the starting point for consideration here of the necessary synchronisation (my emphasis):

> A dependency is a relationship between observables that pertains *in the view of a particular agent*... changes... are *indivisible in the view of the agent.* That is: no action or observation on the part of the agent can take place in a context in which x has changed, but the dependants [targets] of x have yet to be changed.

The above statement represents a significant shift in emphasis in thinking about dependency within the EM literature. Usually (in documentation, teaching and papers), the *guarantees* that the definition concept gives are emphasised. For example, [Yun90, p.27], paraphrased:

> No matter what the values of the source variables are, the value of a definition is *always* equal to its defining expression.

With this usual emphasis, `a is b+c` is interpreted as meaning that:

> "`a` is *always* `b+c`".

The emphasis in DM implementation terms is then on automatic recalculation and propagation of change. A "concurrent update" implementation follows this emphasis: values are always consistent with definition, except when the `update` routine is in progress, when it is not meaningful to examine the state.

With the alternative, more recently topical emphasis, `a is b+c` is interpreted as meaning that:

> "an agent that perceives the dependency `a is b+c` is restricted in some way whenever `a` is *not* `b+c`".

The emphasis in DM implementation terms must then be on indivisibility in states and transitions, and synchronisation between agents. Dependency is created through definition-agent action perceived as indivisible by agents for whom the dependency pertains. Below, I describe my current understanding of this form of synchronisation of concurrent definition-agents.

First, let us define the roles that agents interacting in a definitive system can play. I separate the roles as much as possible into observation of state (O), change of state (C), and update of state (U). A 'U-agent' (shorthand for "an agent playing the U role") is a definition-agent and is distinct (for the purposes here) from a 'C-agent'. The U- and C-agent can be considered to be acting "inside" and "outside" the definition maintainer respectively. Various other separations of the roles are of course possible but these are the ones considered here.

Now we can define some protocols through which to achieve synchronisation. I define an observation to be bounded by 'preO' and 'postO' operations, and a change to be bounded by 'preC' and 'postC'. Following the principles of indivisibility described above, it follows that change must exclude observation until the relevant definitive state has been updated. Therefore we can arrange for the C-agent to invoke the necessary U-agent in an 'invokeU' operation. The U-agent will signal completion of the update with a 'postU' operation. An observation "region" bounded by preO-postO operations must then not overlap with a change-update region bounded by matching preC-postU operations. This type of synchronisation is illustrated for the case of a one-definition script graph in Figure 5.5[7].

The synchronisation described is an instance of the mutual exclusion problem [Dij68]. One way in which this can be solved is through the use of Dijkstra's semaphore primitive. In this single definition script graph case, preO and preC can be implemented as $P(s)$ (potentially causing the agent to be blocked if necessary), and postO and postU as $V(s)$ (causing any waiting agents to be unblocked). Listing 5.1 shows a test implementation of this case, written in the language SR (Synchronizing Resources), "an imperative language for concurrent programming that provides explicit mechanisms for concurrency, communication and synchronisation" [AO93]. The code extends Figure 5.5 slightly by modelling two observing agents, O and O2. The agent O adheres to the interaction protocol, using the semaphore, and will never observe `a`, `b` and `c` in a state where `a` is *not* `b+c`. The agent O2 does not use the semaphore and hence it is possible for that agent to observe state inconsistent with the definition `a is b+c`.

Keeping to a one-definition script graph, but extending the example with more O- and C-agents would lead to a problem approximately[8] equivalent to the classic readers/writers problem [CHP71].

[7]The conventions for the diagram are based loosely on [AO93, p.117].

[8]"Approximately" because it is unclear whether multiple concurrent writers should be allowed.

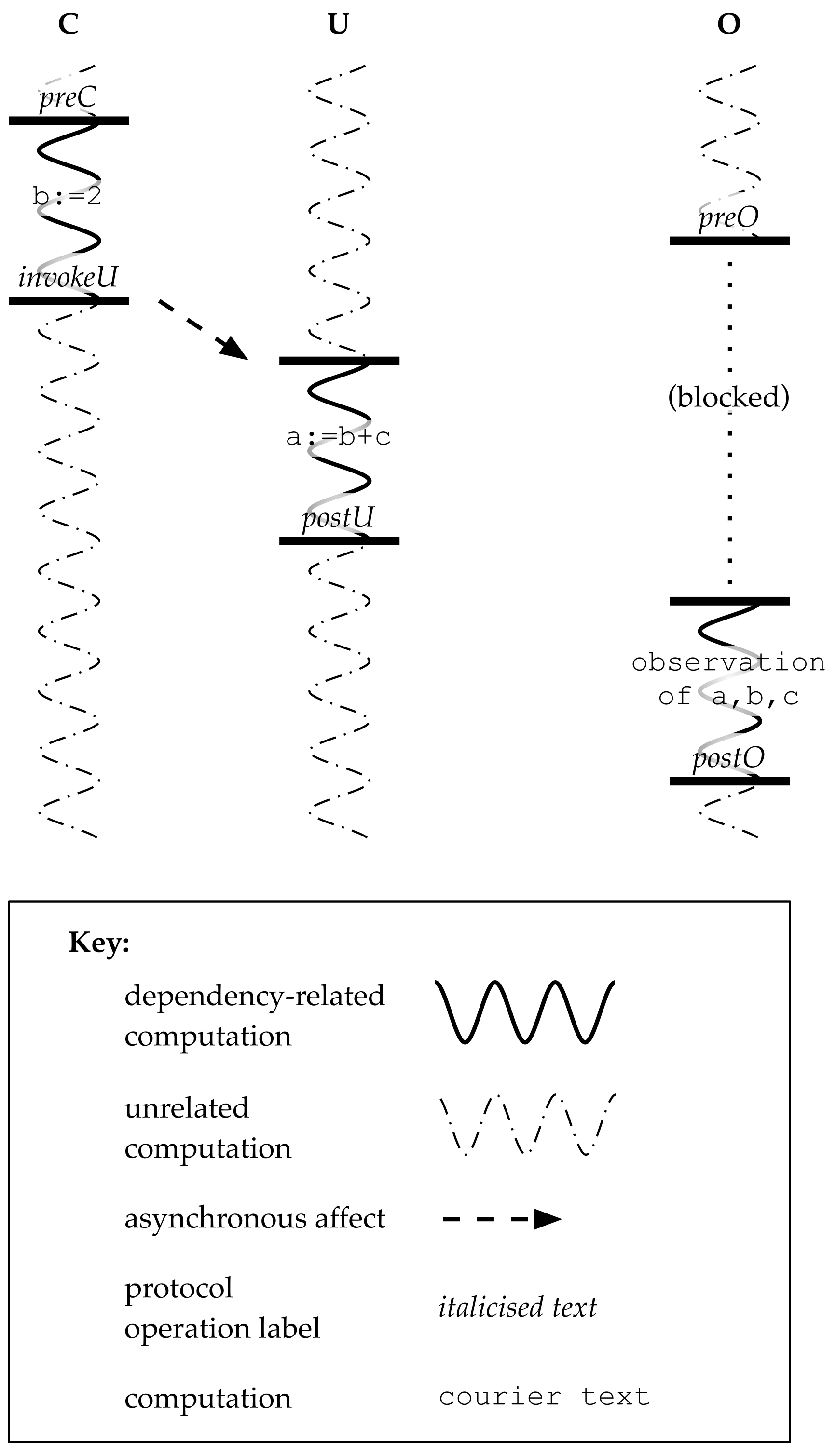


Figure 5.5: Synchronisation for indivisible observation of state in the single definition case

```
resource CUO()

  var a := 3, b := 1, c := 1    # initial values of dependency

  sem s := 1                    # maximum 1 agent acting simultaneously
  op U() {send}                 # U is to be asynchronously invoked

  # O observes a, b and c every so often and perceives a is b+c
  process O
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      P(s)                      # preO: block if C-U is acting
      write("O", a, b, c)       # observe a, b and c together
      V(s)                      # postO: allow C-U to act if blocked
    af
  end O

  # O2 observes a, b and c every so often but does not perceive dependency
  process O2
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      write("O2", a, b, c)      # observe a, b and c together
    af
  end O2

  # C changes b every so often
  process C
    fa i := 1 to 20 ->
      nap(int(random(100)))     # pause to introduce some non-determinism
      P(s)                      # preC: block if O is acting
      b := int(random(20))      # change b
      send U()                  # asynchronously invoke dependency update
    af
  end C

  # U is the agent that updates the dependency
  proc U()
    a := b + c                  # recalculate the dependency
    V(s)                        # postU: allow O or C to act if blocked
  end U

end CUO
```

Listing 5.1: SR code implementing the synchronisation shown in Figure 5.5

Extending the script graph beyond a single definition, to address a more realistic situation, changes the problem significantly. The additional complications relate to simultaneity of observation and change.

1. A definition describes a relationship between *multiple* nodes in the script graph. Simultaneity of observation of nodes and also, simultaneity of change to nodes thus become issues to be addressed. One way this can be added to the framework described so far is to extend the preO and preC operations, requiring information about the identity of the *set* of nodes that the agent is to observe or change.

2. The quote about dependency by Beynon *et al* cited above from [BCSW99] implies that dependency is a form of guarantee: when observed, values in a script graph are guaranteed to be consistent with their definition. The full implications of this statement go further: observation of the value of a node by induction entails observation of all values of the recursive sources of that node.

3. Immediately after a change is made to a value at a node in a script graph, the values of the recursive targets of that node potentially become inconsistent with their definition. Note that changes to the *definition* of a node have the same effect: although changes to a definition change the structure of the script graph, the values of the nodes *beneath* the change are unaffected (although these nodes may gain or lose a target reference).

The above discussion leads directly to a suitable strategy[9] for synchronising observation, change and update of a script graph.

- Observation excludes change to recursive sources *below* the observed set of nodes, whilst observation is in progress.

- Change excludes observation of recursive targets *above* the changed set of nodes, until each value has been updated to be consistent with its definition.

[9]This is only one possible way of implementing the indivisibility requirement and may be unnecessarily restrictive.

Observation and change-update can therefore be thought of as placing restrictive 'curtains' over segments of the script graph. Observation of nodes (signalled by preO) causes a change-restricting curtain to be placed *below* the nodes to be observed. Once the observation within the curtain is complete, the curtain is atomically removed by postO. Change to nodes (signalled by preC) causes an observation-restricting curtain to be placed *above* the nodes to be changed. Once the change within the curtain is complete, the state can then be updated (concurrently where possible) by the U-agents, which signal completion of update of a node with postU. This then removes the restriction on observation. Therefore, as each node within the 'curtain' is updated, the curtain is lifted, progressively revealing the new state.

Appendix §5.A (p.328) shows an SR program that implements the multiple definition case. The program declares one semaphore for each definition in the script graph. The nodes beneath a particular set of nodes can be locked before observation of the set and unlocked afterwards. The nodes above a set of nodes can be locked before the set is changed. The values are then recursively updated, concurrently if a node has more than one target. The locks are removed as the state is updated. Although each node semaphore within a set must be locked sequentially, deadlocks do not occur as each concurrent (possibly competing) process makes the $P()$ and $V()$ operations in the same sequential order.

The "power" script example (Figure 3.2, p.115 *et seq.*) is implemented in the SR program with several concurrent processes that make various redefinitions and observe various node values. Some processes adhere to the interaction protocol and so always observe consistent state, some processes do not and so can observe inconsistent state.

Although the program functions as described, it is a prototype implementation only. Particular problems include that only one O-agent can be active on a set of nodes at any one time (a readers/writers solution would be better) and also the data describing the topology of the script graph (as opposed to the values) is not protected from concurrent update.

The above strategy is not the only possible way of meeting the indivisibility requirement. For example, it might be possible to allow change beneath an observed node, as long as the update that follows is prevented from propagating into the nodes being observed. Another example of a possibly more general model would be as follows. The propagation of update following a change can be considered to form a 'ray' up the graph, originating from the change. Before a node in the path of the change is updated, it is in state S. After the ray has passed, it is in state S'. Observation of a set of nodes is permitted, so long as the entire set is either covered or uncovered by a ray (i.e. the entire set is in state S, or the entire set is in state S'). One complication here is that the covered/uncovered requirement applies only to nodes in the path of a change.

5.2 Moding

5.2.1 Problems with definitive lists

Section §3.5 discussed several problems related to the geometry of the DAM machine definitive store. There are significant problems involved in using data types whose representation is larger than one word, and the extension of the DAM machine to include a list data type poses very large problems. The `lookup` operator proposed by Cartwright ([Car99, p.159] — see §3.5.1) does not implement propagation of change from within the list and implementing this necessary dependency requires a fundamental rethink of the design.

Eden does include a list data type, and it is widely used. (For example, in the current `tkeden` implementation, the DoNaLD translator transforms each DoNaLD object to an Eden list, which is then used by utility routines written in Eden for rendering and other calculations.) However, there are various problems in the current EDEN implementation of lists. When used by procedural Eden code, treating a list as a conventional RWV, the implementation behaves as might be expected. When a list is referenced by a definition or defined as a FV, however, there are problems in every case.

Four problems are shown below to illustrate the issues as they appear in use of the Eden language. In each example, a comment starting with a `!` character indicates a problem.

Problem 1: Observation of a list FV causes evaluation

```
1  %eden
2  l is [1, g(x)]; /* g is undefined */
3  writeln(l[2]);  /* error: func "g" needed */
4  writeln(l[1]);  /* ! error: func "g" needed */
```

In this example, it is not possible to observe the value of the second element of the list (at line 3) as the function g has not yet been defined. Observation of the first element (at line 4) should however be possible, but the current EDEN implementation gives an error. Generally: observation of an individual list element causes the *entire* list to be evaluated.

Problem 2: Difficulty of redefining portions of a list FV

```
1  %eden
2  a = 1; b = 2;
3  l is [a, b];
4  l[1] is 3; /* ! parse error */
5
6  ?l;        /* gives "l is [a,b]" */
7  l[1] = 4;  /* this input is accepted, but now the entirety
8                of l is a RWV, not a FV */
9  ?l;        /* ! gives "l = [4,2]": the dependency between
10               a, b and l has been lost */
11
12 proc p: l { writeln("l has been changed"); }
13 b = 5;     /* ! triggered action p is not invoked */
```

Firstly, line 4 of this example shows that redefinition of part of a list FV is not implemented. Secondly, lines 6–13 show that making a procedural assignment to one element of a list FV makes the entire list a RWV, which is often not what is desired. Generally: it is not possible to redefine portions of a list FV[10].

[10] It is possible through use of the Eden language to process the existing definition as a string, make the necessary changes and re-parse the result using the `execute()` command, but this is inelegant and a fully robust solution is complex.

Problem 3: Indiscriminate change propagation with list FVs

```
1  %eden
2  l = [1, 2];
3
4  head is l[1];
5  proc p: head { writeln("head has been re-evaluated"); }
6
7  l[1] = 3;  /* correct invocation of triggered action p */
8  l[2] = 4;  /* ! incorrect invocation of triggered
9                  action p */
```

Here, the FV `head` is observing the first element of the list. When the first element of the list is changed at line 7, the execution of triggered action `p` reveals that `head` has been correctly re-evaluated. However, when the second element of the list is changed (line 8), the triggered action is again executed. Generally: change to an element of a list causes re-evaluation of dependencies observing *any part* of the list.

Problem 4: “Phantom” graph cycles in list FVs

```
%eden
l is [6, l[1]]; /* ! error: cyclic dependency detected */

l is [6, a];
a is l[1];      /* ! error: cyclic dependency detected */
```

In this final example, two attempts are made to define the second element of the list to be the same as the first element, which is defined to a literal value. There are no cycles in the script graph here if `l[1]` is interpreted as referring to the first element of the list *as an individual*. However, the current EDEN implementation makes no provision for such an interpretation and so detects a graph cycle. Generally: “phantom” graph cycles can be inappropriately detected when references to the individual elements of list FVs are made.

At the root of the four problems is the fact that the current EDEN implements a reference to a list element using a function. An element reference is effectively translated internally[11] to a functional representation:

$$\texttt{v is l[i]} \rightarrow \texttt{v is } f(\texttt{l}, \texttt{i})$$

Statically, this translation is "correct": reference into a list is a mathematical function in the sense that the function f maps the `l` and `i` arguments in a uniform way to the result value `v`:

$$f : (\texttt{l}, \texttt{i}) \ \mapsto \texttt{v}$$

The mapping is consistently applied whenever invoked — in other words, definitions are referentially transparent. However, the functional interpretation of the element reference does not capture one essential ingredient of this kind of reference.

If the functional interpretation is taken, then changes to the arguments of the function imply a possible change to the value of the function. If the offset argument `i` is changed, then certainly the value `v` will reference a different element of `l` and so may change. But if an element of the list argument `l` that is *not* referenced by the current offset argument `i` is changed, then the value `v` will certainly *not* change. The functional interpretation applied in this way is simply not specific enough.

The functional interpretation of reference to an element is at the root of the four problems in the current EDEN implementation described earlier. The general lesson here may be that when considering dependency over composite types, the functional abstraction applied at the composite level is inappropriate, as it does not capture an essential aspect of the nature of reference.

Considering propagation of change rather than functional abstraction has revealed the above problems. Further consideration of the issues of lists in this way leads to the following idea. When a change is made, the old value can, in some situations be re-used in order to assist with calculation of the new value. This would

[11]Although note that this is not a translation involving strings and notation — this occurs at the level of EDEN's virtual machine.

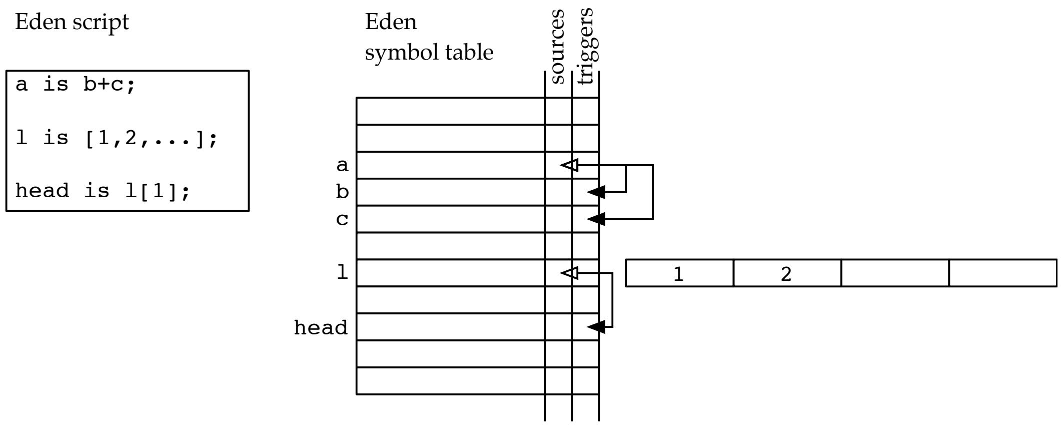


Figure 5.6: EDEN implements lists "horizontally" in the symbol table

improve update efficiency and gives no semantic problems. For example, the sum of a list can be re-used if one element changes:

$$\begin{aligned}\text{composite\_sum\_new} &= \text{composite\_sum\_old} + \\ &\quad (\text{changed\_element\_new\_value} - \text{changed\_element\_old\_value})\end{aligned}$$

As it is the references to composite data structures in EDEN that lead to problems of specificity of reference, one way to work around the problem is to eliminate such data structures. I have constructed a solution in EDEN which involves reducing all composite data structures to multiple atomic variables. In terms of the EDEN symbol table, the problems outlined in this section occur because lists are implemented "horizontally" — all the elements of the list are located conceptually at the location of the list identifier, as illustrated in Figure 5.6.

A macro translator, named the Eden "Symbol Lists" translator was written for EDEN in Eden, using the regular expression facilities and generalised notation framework (described in §4.2.4). The translator performs macro transformations on Eden input before it is parsed, changing language constructions involving lists into a form where the individual elements of the list are each stored in their own symbol. It is activated by changing the notation context to `%edensl`. The component parts of a list '`l`' with three elements will be stored in the symbols `l1`, `l2` and `l3`. The length of the list is stored in another symbol, `ll`. The exact transformations used are illustrated in Figure 5.7.

```
l = [a, b, 42];  → edensl_assignconstruct("l", [a, b, 42...]);
l is [a, b, 42];  → edensl_defineconstruct("l", ["a", "b", "42"...]);
l[4] = a;  → edensl_assignelement("l", 4, a);
l[4] is a;  → edensl_defineelement("l", 4, "a");
l#  → ll
l[i] (on RHS)  → li
append l, v;  → edensl_append("l", v);
delete l, i;  → edensl_delete("l", i);
insert l, i, v;  → edensl_insert("l", i, v);
shift l;  → edensl_shift("l");
l = ...  → edensl_assign("l", ...);
l is ...  → left unchanged
```

Figure 5.7: Transformations implemented by the `%edensl` translator

As the macro transformation causes the list components to be stored separately, after transformation, there is no way of observing the list as a whole. This problem is solved by the introduction of a definition with a name as in the original construction, defined to be a list FV, naming each individual element as a component. This definition needs reconstruction when the list changes in length, and this is done by a triggered action. Figure 5.8 shows an example transformation and a representation of the resulting symbol table, where it can be seen that the list is now stored "vertically", each element in an individual symbol.

The `%edensl` macro translator solves all four problems mentioned earlier. It can even be configured to "replace" the standard `%eden` notation, in which role it is transparent to the user. (The original Eden is still available by using the notation context `%eden0`, as illustrated in the script in Figure 5.8.) The decrease in performance that the macro translator causes may actually be quite small, since once the transformation and parsing is complete, EDEN stores the virtual machine opcodes — re-parsing is not necessary during machine execution. However, a major limitation of the macro translator is that the blocking of the regular expression transformations is on a per-line basis, and the EDEN `execute()` routine used to pass transformed output to the Eden interpreter does not accept partial input. The translator therefore currently fails to process multi-line procedures correctly.

Script

```
%edensl
l is [a,b,3];
head is l[1];

  ->
    %eden0
    l1 is a;
    l2 is b;
    l3 is 3;
    ll = 3;
    l is [l1, l2, l3];
    proc l_constructwhole : ll {
      /* constructs new definition of l
          when ll changes... */
    }

    head is l1;
```

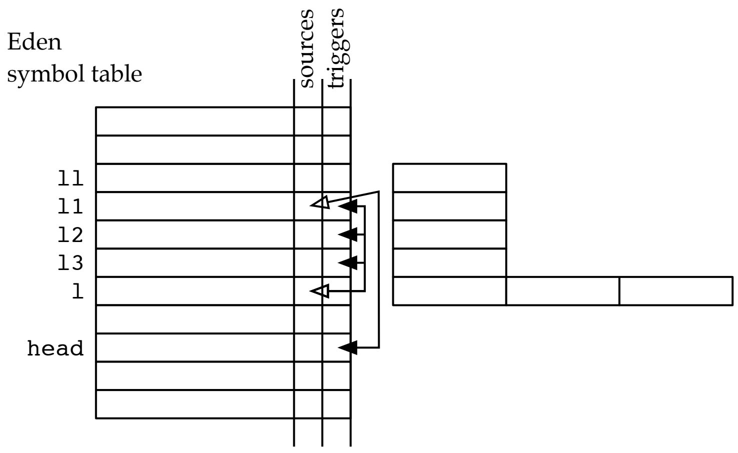


Figure 5.8: The `%edensl` translator transforms references to lists, causing EDEN to locate them "vertically"

5.2.2 Meziani's DENOTA and the "mode of definition" of a variable

Contemporary with the initial development of EDEN, DENOTA (for DEfinitive NOTAtions) was developed by Samia Meziani, who described it in her MSc thesis [Mez87]. The aim was to implement a tool to handle data abstraction within the definitive programming paradigm. Of interest in this section is the concept of *mode of a variable*. The concept had some precedent in Beynon's ARCA notation, first described in [Bey83] (which is the first formal publication about definitive concepts, even predating that term). However, [Mez87, §4] is the only publication with a significant treatment of the "moding" topic[12].

"Moding" can be motivated as follows. In interaction using a definitive notation that supports lists, a variable can be defined to be a function that evaluates to a list, or it can be defined using a list constructor.

After a variable has been defined functionally, later in the interaction, the variable can be redefined. However, later in the interaction, the *components* of the variable cannot be redefined. As is illustrated by the Eden example associated with Problem 2 in §5.2.1, partial redefinition is not possible in general, since (as we show below) such a redefinition would correspond to a "reprogramming" of part of the function in some way.

After a variable has been defined using a list constructor, later in the interaction, similarly, the variable can be redefined. This time, the components of the variable *can* be redefined: partial redefinition is possible. Such a redefinition does not constitute a "reprogramming" of the constructor — the constructor has been used to create the initial "shape" of the list and now the shape, not the constructor, remains.

The following examples are intended to illustrate this point. In order to give meaning to the examples, I have used the Eden language[13]. However, the examples use the Eden syntax as a language independently of our current implementation — as described in the previous section, the current EDEN implementation does not deal well with definitive lists. Also note in particular that this discussion relates

[12][Geh95, Bir91, Car99] variously describe moding but do not add any information beyond that given in [Mez87].

[13]The examples are my own — [Mez87] does not illustrate this basic point with an example and the notation used there appears to be based partially on LISP.

only to definitions: the examples use Eden FVs exclusively — RWVs, created by assignment, do not appear.

First, we illustrate a functional definition and the impracticality of then interactively redefining it component-wise.

```
func f { return [1,2,3]; }
l is f();
l[1] is 4; /* ! not possible */
```

The redefinition of the first component of the variable `l` could imply a redefinition of the function `f`, perhaps to the following.

```
func f { return [4,2,3]; }
```

In the more general case (for example if control flow is used within function `f`) then this modification is not possible to determine automatically.

Alternatively, the redefinition could imply the use of an additional function layered on top of the existing function `f`, modifying the first component, as follows.

```
func g { q = f(); q[1] = 4; return q; }
l is g();
```

However, successive uses of such redefinitions to one variable interpreted in this way would lead to many `g`-style functions building up in the state, effectively representing the entire history of the interaction with that variable.

Use of a list constructor implies no such problems:

```
m is [a, 2, c];
m[1] is 4; /* OK */
```

Given that the variable `m` has been initially defined using a constructor, the redefinition of the first component of `m` implies the following redefinition:

```
m is [4, 2, c];
```

which takes the history of the interaction into account but the result of which is still meaningful statically.

The above discussion exposes serious problems in interpreting the way in which definitions match variables to formulae. Meziani's proposed solution to this problem is to introduce an auxiliary definitive notation in which the mode of definition of variables can be declared. In effect, definitive principles are being used to supply the meta information needed to disambiguate reference and constrain redefinition. A significant aspect of using an auxiliary definitive script to define mode is that the relationship between variables and their definitions can be as flexible and dynamic as value definition in a definitive script.

The application of auxiliary definitive notations of this nature is not confined to handling the mode of definition of variables. For instance, Meziani also proposed that similar principles could be adapted to provide information hiding in definitive scripts [Mez87, p.75]:

> Definitive notations could be augmented to support information hiding. For this, two facilities could be used, namely a reference moding, and a definition moding of variables. The latter would describe the way in which variables are referenced and defined. The former could describe the way definitive variables could be viewed. For instance, a variable of reference mode `abst list` could have hidden components. These concepts would also be useful to express semantic rules such as: the `tail` could apply on `abst list` variables. These potential facilities would provide stronger typing of expression analogous to Abstract data types, and Object-oriented programming paradigms.

Here, moding has a more generic meaning than has been applied so far, since it can be used separately for "reference moding" and "definition moding".

Our current techniques for implementing definitive notations are not sophisticated enough to support such features.

5.3 The tri-box framework for higher-order definitive state

This section sketches a "tri-box framework" for higher-order definitive state that I have developed in response to some of the issues raised in this thesis. The first subsection briefly summarises the motivations for the framework, the second subsection sketches the framework concept and gives some examples and the final subsection outlines issues for implementation raised by the framework.

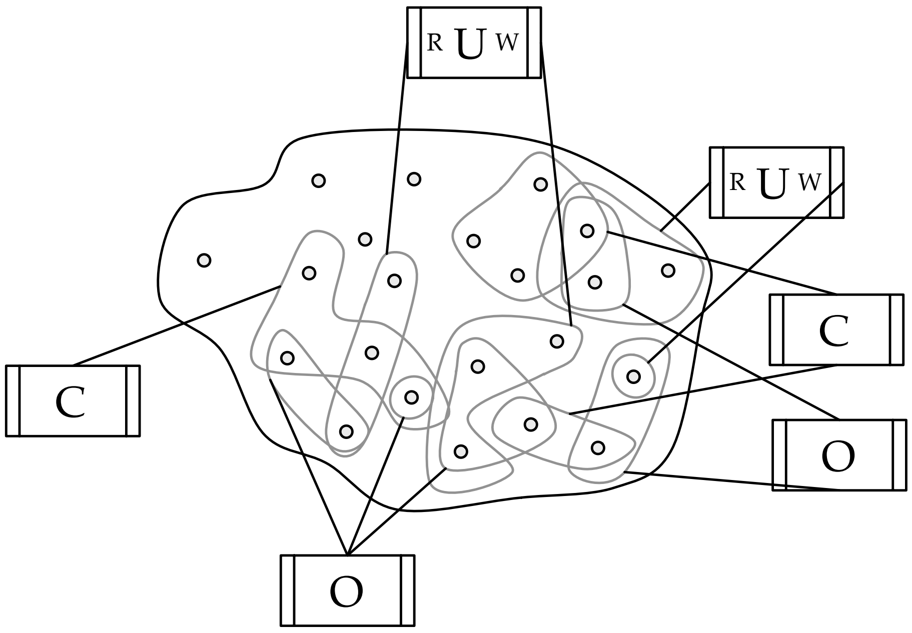


Figure 5.9: O-, C- and U-agents interacting with the state S

5.3.1 Motivation

Section §5.1.2 proposed three roles for agents interacting in a definitive system, in order to consider the types of synchronisation that a concurrent definition maintainer might use. Each of the O-, C-, and U-agent roles (for observation, change and update respectively) observes or acts on a subset of the state within the system. Here, we name the state S (following the discussion of transitions from an initial state S through intermediate $S*$ states to a resultant S' state in §2.3.3). In an LSD account, each agent *perceives* values — there may arguably be no objective state S. In this section, we assume that the authentic values of observables can be directly acted upon or observed by O-, C- and U-agents: the framework is similar to the ADM in this respect. Figure 5.9 illustrates several agents playing different roles interacting with the state S.

In the EM group, we conventionally describe the state S by the use of a definitive script written in a definitive notation. However, much of the work described in this thesis hints that a symbolic definitive script is in many ways an inadequate framework in which to discuss the propagation of change in its full richness. For example, previous sections have made the following points about symbols in a definitive state.

- A definition-agent writes to only one symbol (§4.3.4, §4.3.5). This constraint on definition-agent action gives traceable meaning to the state.
- But a U-agent in the extended script digraph can write to more than one symbol (§5.1.1). This can be necessary for implementation efficiency reasons with current machine architectures, where there is not enough parallelism available to make a one-to-one mapping of U-agents to the script graph. It may also be necessary for semantic reasons in the absence of notations describing powerful higher-order dependency, which could for example be used to describe the dependency involved in updating the screen (§3.5.4).
- The mapping of a symbol to locations of words in store is a complex distraction at the level considered here. One symbol can correspond to more than one word in store, depending upon type (§3.5.1). The symbol table is another level of indirection (§3.4.3), itself a higher-order dependency. We wish to use dependency in the symbol table to create dependency-driven parsers (§4.2.5).
- The functional abstraction of reference implied by symbols in a definitive script causes many problems in the use of definitive lists (§5.2.1).
- The dependency described by a definition need not be objectively perceived — different agents may have different perceptions (§5.1.2).

The presence of symbols implies that there is one objective understanding and state of each symbol. However, in our natural language we use symbols for *reference* only and in many instances we start from an assumption of subjective understanding.

Are there atoms in definitive state at all? We may wish to introduce variously-sized atoms in order to cope with different data types (§3.5.1); atoms that have hidden composite structure (one particular mode of definition — §5.2.2); structures formed from atoms (another mode of definition), perhaps with some values hidden from certain observers (as Meziani suggests in her "reference moding" concept). If there are atoms, are they ordered in any way? We may wish for various forms of ordering: words in store can be considered to be ordered by a single dimension;

the screen naturally has a two dimensional ordering (§3.5.4); spreadsheets have an ordering that is at least two-dimensional (and possibly three-dimensional, if for example Excel's 'sheets' are taken into account). Pursuing the connections developed in [Bey97, Bey99, Bey03], the nature of definitive state may be likened to that of William James's "unfinished pluralistic universe"[14]. Notice that Figure 5.9 illustrated S as an unordered set of atoms of which there was a single basic type.

In order to tackle some of the above issues, this section moves away from the notion of the symbolic definitive script. Instead, below, we take S to be a sequence of atomic 'boxes' in store.

The framework described below differs from the DAM machine as described in Chapter 3. Although the DAM machine may appear to have been designed with a primary focus upon dependency between atomic words of store, it is actually an 'implementation' of the DMM. The DMM is a formalisation of the Low Level Definitive Notation (LLDN) concept, a symbolic notation describing dependency between a set of atomic integer values, and hence the symbolic influence on the DAM machine design is strong. LLDN can describe dependency only between single words in store — neither data types larger than a word (§3.5.1) nor lists (§3.5.2) are possible. As the basis of LLDN is a *set* of integer values, identities are not structured in the DAM machine, as they would be in an authentic "generalised spreadsheet" (a term previously used to describe definitive scripts — see §3.5).

The focus on the sequence of atomic 'boxes' in store below also means that we must treat the concept of the script graph with caution. A script digraph (see Appendix §3.A, p.178) describes the propagation of change required in the symbolic script structure. An extended script graph describes how the propagation of change is mapped to updating agents. But if we wish to consider higher-order dependency, the script graph is itself subject to dependent change[15]. The change may be limited to just the arcs in the graph (for example, in the case of the `if` HOD, described in §4.3.7) or it may also involve the addition or removal of nodes (for example, in the

[14]Wild describes this universe as having "...aspects of unity, relations which hold different members of this collection together. But there are also aspects of diversity and independence. As we live through this empirical world, it is 'like one of those dried human heads with which the Dyaks of Borneo deck their lodges. The skull forms a solid nucleus; but the innumerable feathers, leaves, strings, beads, and loose appendices of every description float and dangle from it, and, save that they terminate in it, seem to have nothing to do with one another' [Jam12]" [Wil69, p.391].

[15]This thesis is seemingly the first writing to describe Higher-Order Dependency in this way.

case of ADM entities instantiated or deleted by the redefinition of a `LIVE` variable, mentioned in §2.1.1).

By moving away from the notion of the symbolic definitive script and taking S to be sequence of atomic 'boxes' in store, we can concentrate in this section on the synchronisation of mechanism and perception involved in interaction with meaningful state. The mechanism here is the action of U-agents in response to change initiated by C-agents. The perception is achieved by synchronisation of O-agents with respect to C-U-agent action. O-, C- and U-agents all act on or observe a subset of S. The desired synchronisation and interaction with subsets of S is illustrated in Figure 5.10.

The tri-box framework does not attempt to solve all these issues. However it does seem to clarify:

- the concept of U-agents that write to more than one atom (for example, the screen);
- the problem of indiscriminate change propagation associated with definitive lists (Problem 3 described in §5.2.1);
- restricted forms of higher-order dependency (involving dynamic script graph arcs), and
- the synchronisation required for subjective dependency.

The tri-box framework focusses on *propagation of change* within the current state, rather than *evaluation* of a script. The framework has emerged by asking the questions: *What is the minimum information we need to implement a concurrent definition maintainer?* and *What is the simplest way to organise the information?* In asking such questions, we have moved away from the symbolic emphasis of scripts, but hope to return with some insights for our symbolic notations.

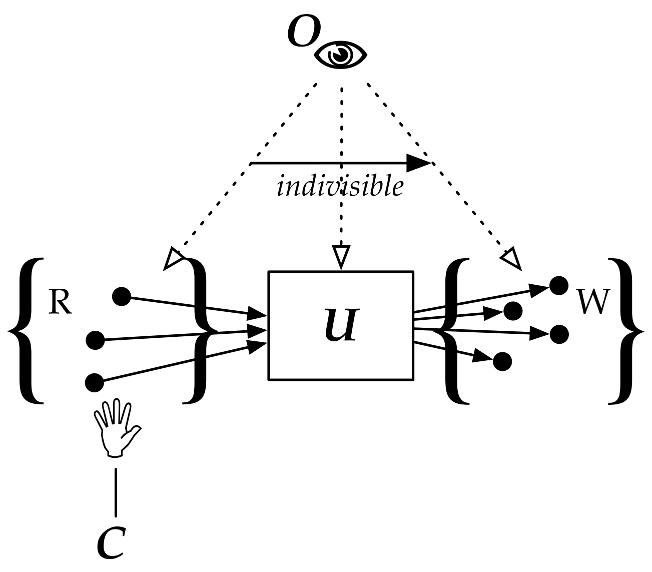


Figure 5.10: O-, C- and U-agents observe & act on subsets of S and are synchronised

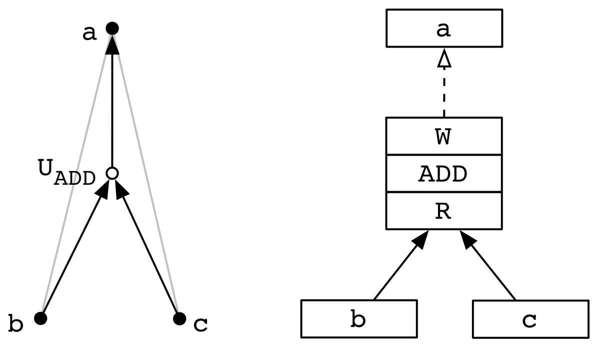


Figure 5.11: Script graph and tri-box diagram for `a is b+c`

5.3.2 Concept and examples

The tri-box framework is a “visual formalism” [Har88] describing the propagation of change within a sequence of atomic ‘boxes’ in store. It can be viewed as a generalisation of the extended script graph (described in §5.1.1).

The extended script digraph for the definition `a is b+c` and the corresponding tri-box diagram are shown in Figure 5.11. The value nodes of the extended script digraph have been replaced by ‘value boxes’, each holding a value in the store S. A value box may or may not correspond to a word in store and may be of fixed or variable length — these are matters of implementation.

The U-agent node in the extended script digraph has been replaced by three vertically joined boxes, termed a ‘tri-box’. From top to bottom, the three boxes in a tri-box are named the ‘W-box’, the ‘U-box’ and the ‘R-box’. The values contained in the boxes describe the ‘W-set’, the identity of the update operator and the ‘R-set’ respectively. The W-set is the particular subset of boxes in the store S that the U-agent may write values to, and similarly the R-set is the subset of boxes that it may read values from. The subset of S referenced by each U-agent can therefore be specific to each U-agent. The framework captures the notion of subjective reference as opposed to the identification of objective symbols.

The value of the W-set is held conceptually in the (top-most) W-box, and is diagrammatically represented by drawing ‘W-coloured’ arcs from the W-box to the boxes included in the W-set. Similarly, the value of the R-set is held conceptually in the (lowest) R-box. The value of the R-set is represented diagrammatically by drawing 'R-coloured' arcs *from* the boxes included in the R-set to the R-set box. The three adjacent boxes and arcs are drawn this way so as to mirror the conventional geometric layout of the corresponding script graph, where sources of the U-agent are conventionally placed below, targets above, and arcs describe propagation of change.

The dependency in the system is thus described by the information held collectively in all the tri-boxes. Following Slade [Sla90], this information is known as D (see §2.1.2). The information is used to coordinate and synchronise the O-, C- and U-agents, as outlined in §5.1.2 and illustrated in Figure 5.10.

If the boxes making up the tri-boxes are located in S along with the 'value boxes' (this organisation can be succinctly termed 'D-in-S'), the values contained in the tri-boxes may be written to by U-agents. In this case, the script graph itself is maintainable by dependency. The framework therefore provides a conceptual means with which to describe higher-order dependency.

There is one rule restricting the topology of a tri-box diagram, which follows from the discussion in §4.3.4 of constraints upon definition-agent action:

> Each box in S must be referenced by *at most one* W-set.

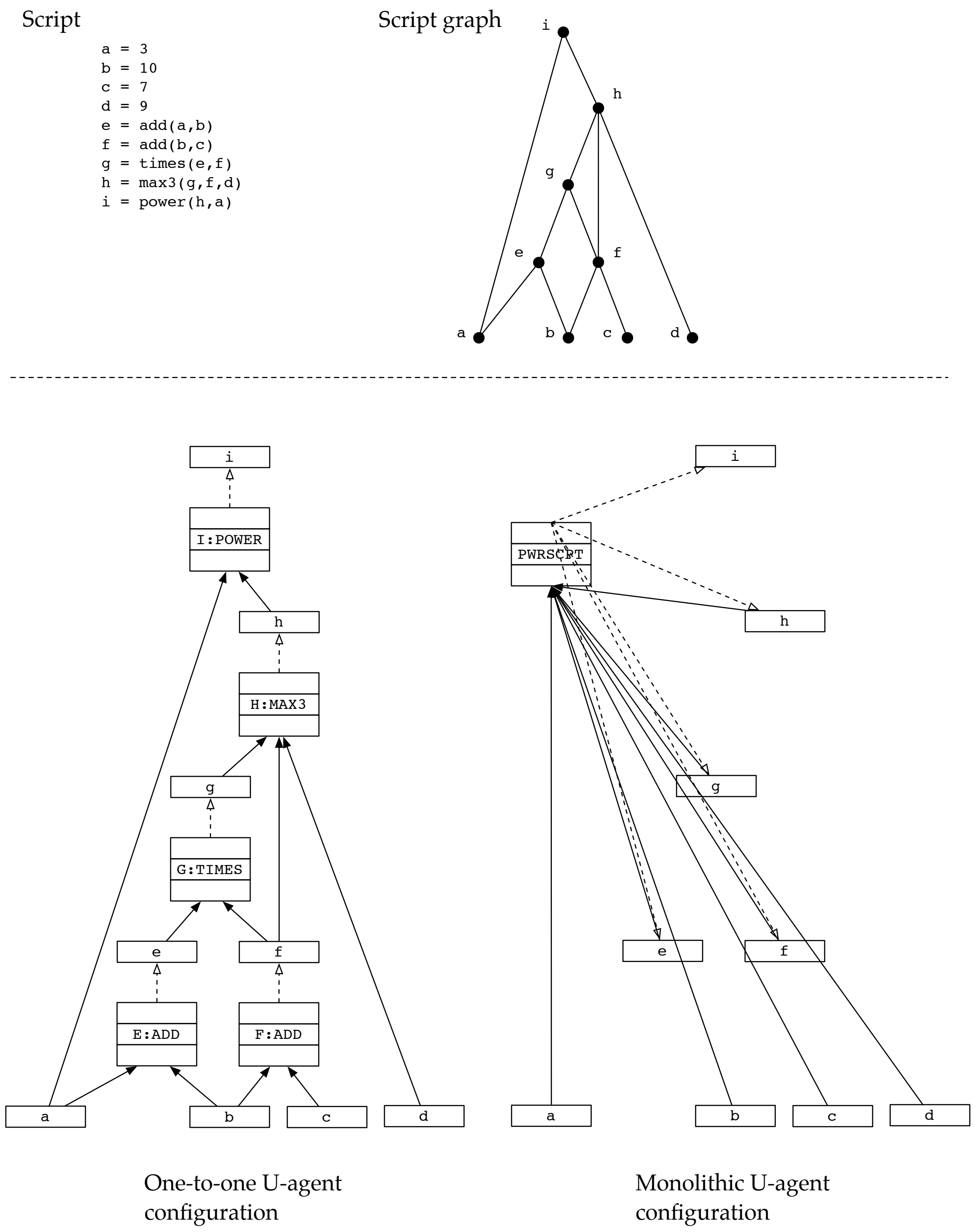
Alternatively stated, considering the W-sets as sets of arcs (as drawn in Figure 5.11), each box in S must have at most one incoming 'W-arc'. This rule ensures that no word in S is subject to change from two or more independent agents. It is then possible to trace effect back to cause.

Some examples to illustrate how the tri-box conceptual framework can be applied are given on the following pages. The next subsection then discusses implementation of the framework.

Example 1: The "power" script — illustrating many possible U-agent configurations

Two non-trivial examples of the tri-box diagrammatic form are shown in Figure 5.12, which depicts two configurations for maintaining dependency in the "power" script (a running example in this thesis, which first appears in §3.2.4). The figure shows the "power" script, the script graph and two possible tri-box implementations of the script graph. The tri-box implementation shown on the left has U-agents in one-to-one correspondence with nodes (the "one-to-one configuration" — see §5.1.1). Notice that there are two distinct `ADD` update operators, for reasons discussed further in the next subsection. The tri-box implementation shown on the right uses a single U-agent. This is a 'monolithic' configuration (see §5.1.1).

The tri-box framework can represent the many possible mappings of script graph nodes to updating agents which are possible between these two configuration extremes (*cf.* the discussion of extended script graphs in §5.1.1). Each possible mapping has differing potential for concurrent update.



Script

```
a = 3
b = 10
c = 7
d = 9
e = add(a,b)
f = add(b,c)
g = times(e,f)
h = max3(g,f,d)
i = power(h,a)
```

Script graph
i
h
g
e
f
a
b
c
d
I:POWER
H:MAX3
G:TIMES
E:ADD
F:ADD
PWRSCPT
One-to-one U-agent configuration
Monolithic U-agent configuration

Figure 5.12: Tri-box diagrams of the “power” script

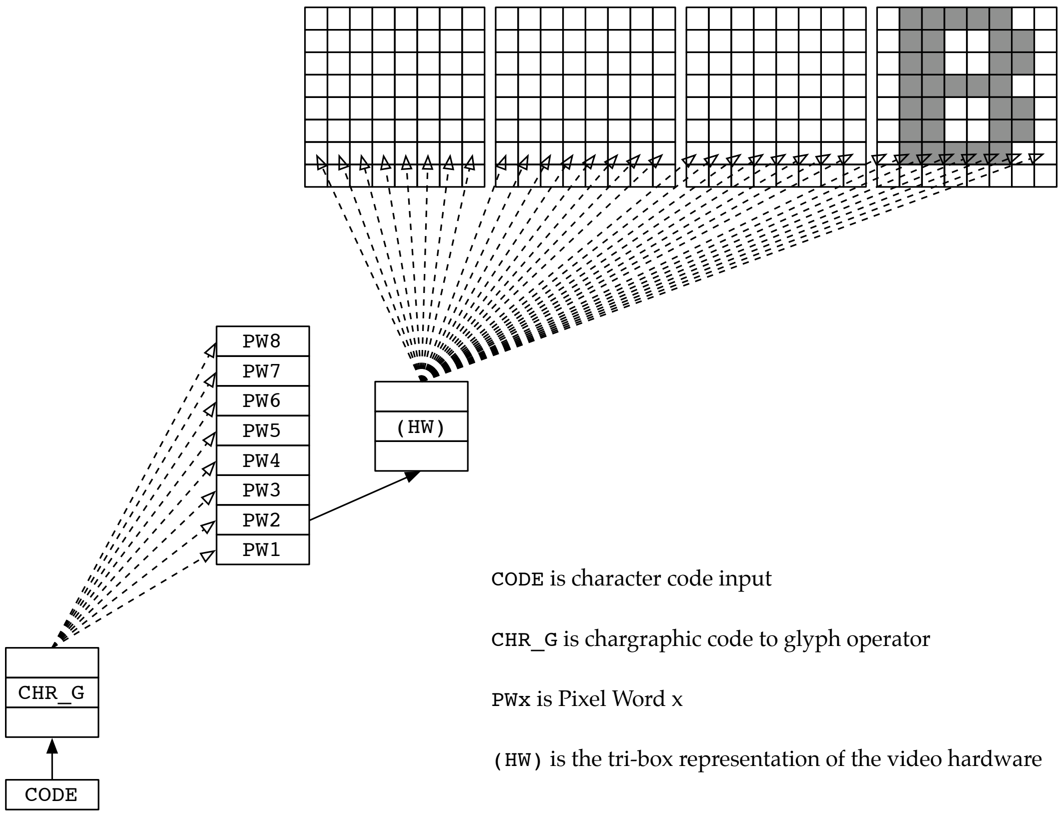
Example 2: Character glyphs — multiple W-set items and hardware

This example demonstrates some of the potential for using W-sets containing more than one reference, and the ability of the framework to represent dependency that is present externally to a definition maintainer. The example is related to the one given in §3.5.4, where a character pattern dependent on a character code word was made to appear on the screen.

At the bottom left corner of Figure 5.13 a single U-agent with the update operator `CHR_G` is shown. This U-agent reads from a single value box containing a character code (labelled `CODE`). The `CHR_G` ("character code to glyph") operator then internally calculates the appropriate character glyph and writes to eight value boxes (labelled `PW1` to `PW8`).

The configuration of the DAM machine described in §3.5.4 exploited the video hardware of the machine to directly render definitive state. In this configuration, the video hardware can be considered to be a U-agent, as on-screen state is indivisibly (at the level of human perception) related to video RAM state. This U-agent can be represented as a tri-box as shown in the figure, where the "update operator" is labelled '`(HW)`'. As shown in the figure, the single box `PW2` (in implementation, a 32-bit machine word) corresponds to 32 on-screen pixels[16]. Only one box to pixel region mapping is shown for clarity. Multiple mappings could be achieved by adding more tri-boxes or by extending the R- and W-sets of the existing '`(HW)`' tri-box.

[16]In the black and white graphics mode used by the DAM machine in single-tasking mode, one 32-bit machine word corresponds to 32 on-screen pixels, each bit representing one binary pixel state.



`CODE` is character code input

`CHR_G` is chargraphic code to glyph operator

`PWx` is Pixel Word x

`(HW)` is the tri-box representation of the video hardware

Figure 5.13: Tri-box diagram showing an on-screen character glyph representation linked by dependency to a box containing a character code

Example 3: Observation of overlapping subsets — discriminate references

This example relates to the problem of indiscriminate change propagation in EDEN, discussed previously as Problem 3 in §5.2.1. In the present EDEN, change to any element of a list causes re-evaluation of dependencies observing *any part* of the list, whether the changed element is observed or not. The problem is due to the reliance in Eden on a single symbol to represent the entire list, and the functional abstraction of reference employed, such that the list element `l[1]` is represented internally as the functional $f(\texttt{l}, \texttt{1})$.

In the tri-box framework, references (to subsets of S) are specific to each U-agent — they are not objectively encapsulated in a symbol. Figure 5.14 shows two U-agents observing overlapping subsets of S. Due to the specificity of reference of the R-set, there is no need for the implementation to invoke the updating agent `U2` when a value box not contained in its R-set (such as the box marked `T`) is changed.

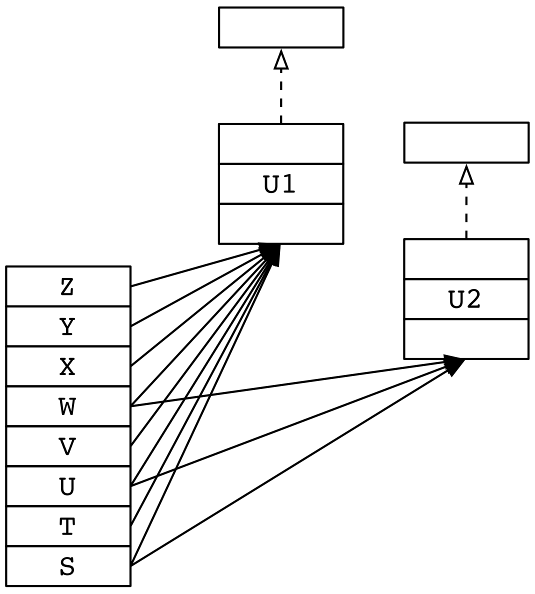


Figure 5.14: Two U-agents observing sub-sets of a list

Example 4: The `if` operator — Higher-Order Dependency

Section §4.3.7 briefly described the Eden ternary `if` operator and problems with its use in a FV due to the functional abstraction of reference. Effectively, the Eden definition:

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

is represented internally as the functional:

```
v is f(b,x,y);
```

but a representation correctly reflecting the propagation of change required would, depending upon the value of `b`, list *either* `x` or `y` as a source of `v`, but not both. The presence of both `x` and `y` as sources in the current implementation can cause "phantom" graph cycles to be detected (see Problem 4 in §5.2.1).

The ternary `if` creates a simple higher-order definition when used in a formula[17]. In this example, the value of `b` affects the arcs required in the script graph. Considered this way, the script graph arcs can be reconfigured when necessary using the following Eden triggered action, presented in §4.3.7:

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

The same solution can be modelled in the tri-box framework if we locate D-in-S. The W-set of the `WRITEREF` tri-box in Figure 5.15 references the R-box of the `COPY` tri-box. The `COPY` update operator implements the simple identity function: the `COPY` tri-box simply copies the value referenced by its present R-set to the value box `v`. The R-set of the `COPY` tri-box is made to reference either `x` or `y` by the `WRITEREF` tri-box (the alternative possibility to the current state is denoted by the grey line drawn from the `y` value box). The `WRITEREF` update operator has as output the value of the R-box representation denoting either a reference to the value box containing `x` or that containing `y`, the choice of output depending upon the boolean value of the input `b`.

[17]Although not when used in an assignment, since this does not create a definition.

The tri-box solution to this problem shown in Figure 5.15 can in principle be examined statically to trace cause and effect. There is an answer to the question "Why is the value of `v` currently that of `x`?" An answer to this question in the Eden triggered action solution depends upon the implementation making a record of the identity of the last action to change the definition of `v`.

The `if` HOD example involves reconfiguration of only arcs in the script graph (R-sets in a one-to-one tri-box configuration). It may also be possible to use the tri-box framework to describe more complex higher-order dependency, involving the creation or removal of script graph nodes under dependency control. To implement this would require a U-agent able to create entire tri-boxes somewhere in D. A dependency-driven parser would require such a U-agent.

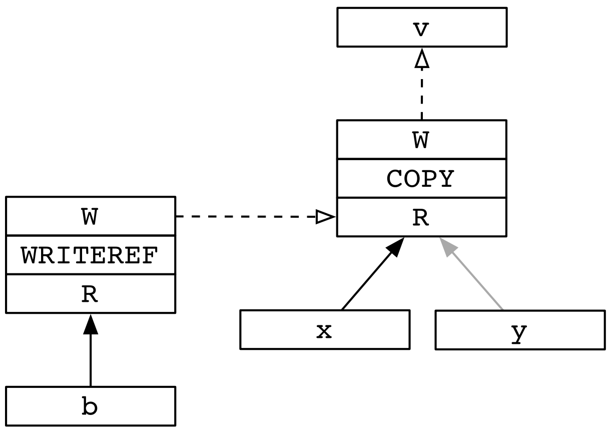


Figure 5.15: An `if` conditional HOD in the tri-box framework

Example 5: Subjective dependency

The characterisation of dependency from Beynon *et al* [BCSW99] quoted in §5.1.2 emphasises the *subjective* character of the dependency concept: “A dependency is a relationship between observables that pertains *in the view of a particular agent*” (my emphasis). Section §5.1.2 contains an example implemented in SR where two agents, `O` and `O2`, observe the three variables `a`, `b` and `c`. The agent `O` perceives the dependency relationship ‘`a is b+c`’ in the state, but the agent `O2` perceives only unrelated variables. It is possible to describe such ‘subjective dependencies’ by adding observing agents to a tri-box diagram.

Figure 5.16 shows two O-agents observing five value boxes. The agent O1 perceives only the dependency described by the `C:ADD` tri-box, and the agent O2 only that described by the `E:ADD` tri-box. As a result, whenever the agent O1 makes an observation, the state will be consistent with the relationship ‘`c is a+b`’. However, the state may not be consistent with the relationship ‘`e is c+d`’ — the agent O1 is able to observe the state during the time period between the start of a change to the value of `c` and the end of the execution of the update operator `E:ADD`. Conversely, the agent O2 will always perceive state to be consistent with ‘`e is c+d`’, but not necessarily ‘`c is a+b`’.

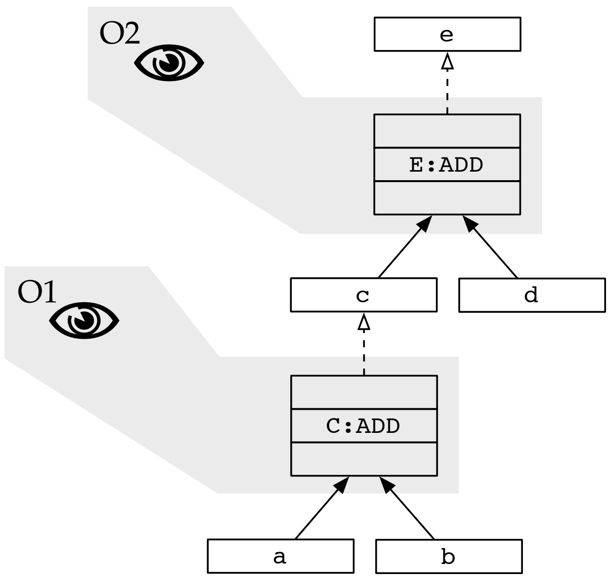


Figure 5.16: Two O-agents perceiving different dependencies

5.3.3 Implementation issues

The tri-box framework is a useful conceptual framework for understanding and describing the issues outlined in the previous section relating to the implementation of dependency. The framework provides a conceptual guide for implementation but it is not a detailed precise specification. Many questions remain to be answered before a full concurrent definition maintainer implementation can be considered. This section briefly outlines some of the issues for implementation that the framework raises by considering each example presented in the previous section in turn.

Example 1: The "power" script

A tri-box diagram can be implemented in many different ways. Each implementation requires decisions to be made about data structures and operational organisation.

The DAM machine data structure for the "power" script is shown in Figure 3.2 (p.115). In this implementation, two linked lists are attached to each value in the definitive store, one containing source and the other containing target pointers to other value locations. Note that in Figure 3.2, the Targets Store encodes the same information as the Sources Store. The two stores are used to improve the efficiency of change propagation, unlike the R- and W-sets of the tri-box framework, which do not hold redundant information.

A design using lists of source and target pointers works well if the number of arcs per script graph node is small. If this assumption is false (for example, if the source values for a definition are the entire screen state), then the lists may become unmanageably long.

In operation, use of the DAM machine involves the queuing of redefinitions which are then processed when an `update` routine is invoked. Considered at this level, the DAM machine is always in one of the two states — QUEUING or UPDATING. In terms of concurrency, the DAM machine therefore implements a 'monolithic' U-agent configuration, as shown in Figure 5.12. A concurrent definition maintainer design might implement a one-to-one U-agent configuration as shown in Figure 5.12.

One simple way to evaluate the design of a concurrent definition maintainer is to consider how it acts on two disjoint subgraphs of a script graph — for example, two copies of the "power" script. In a truly concurrent design, the synchronisation for each subgraph should be independent: O-, C- and U-agents observing and acting on one subgraph should not be affected by other agents observing and acting on the other. Such a design is required to have no centralised data or entity which could reduce the independence of agents observing and acting upon subgraphs. This chapter has suggested that the primary parts of a concurrent definition maintainer are the data structure and protocols for concurrent interaction with that data structure — there can be no centralised single-threaded 'definition maintainer' code.

The "power" example illustrates the implications of the rule restricting the topology of a tri-box diagram: each box in S must be referenced by at most one W-set. Figure 5.12 (p.312) follows this rule faithfully. It should be noted that the restriction also applies to user input: in Figure 5.12, the user may not directly change the values contained in the value boxes `e`, `f`, `g`, `h` or `i`. We may assume that every box without an incoming W-arc implicitly has an incoming W-arc representing the agency of the user. The user may thus change the values contained in the 'leaf' value boxes `a`, `b`, `c` and `d`, and also the values contained in the tri-boxes. The restriction has much relevance to the topic of moding discussed in §5.2.2.

The final aspect of the "power" script to be raised here concerns the "power" function itself. Notice that, in the "power" script, `power()` is the only function which is not symmetric in its arguments (since $a + b = b + a$ etc, but $x^y \neq y^x$). In the one-to-one configuration in Figure 5.12, it is appropriate for every other tri-box to reference an R-*set*, but the `I:POWER` update operator requires more information which could be described as an R-*sequence*. The tri-box framework is not based upon R- and W-sequences however — the framework is intended to describe the propagation of change and therefore sequencing of operator invocation. Sequencing of operator *arguments* is a detail that is not relevant at this level, and so the tri-box framework leaves this for the implementation. It is noteworthy here that the EDEN implementation creates a custom VM code 'operator' for each definition at parse time, encoding the references to operator arguments in the VM code. The EDEN scheduler then invokes the VM code at the scheduled time, without concern for the sequencing of operator arguments.

Example 2: Character glyphs

When discussing Example 1 above, we considered the scenario of a single definition having the entire screen state as its source value. Although this is possible in principle in the DAM machine, it would create an unmanageably long source pointer list and many short target pointer lists. Example 2 involves the dual situation, showing eight values (`PW1`...`PW8`) that each have a single source value (`CODE`). In the DAM machine, this aspect of Example 2 would require eight separate definitions, since in that design, operators may only return a single value. In contrast, in the tri-box framework, U-agent operators may write to many values. This example can be constructed as a single U-agent, or as eight U-agents (each writing to a single value), depending upon which is more appropriate.

In the tri-box framework, values are firstly separated from definitions, through the concepts of the 'value box' and the 'tri-box'. Values and definitions are joined again through the use of the R- and W-set concepts, which, if D-in-S is countenanced, also allows the consideration of higher-order dependency, when the script graph information held in tri-boxes can also be used as value boxes.

The tri-box framework is more general than the DAM machine design, and also, in fact, more general than the concept of the LLDN upon which the DAM machine design is based. The tri-box framework may therefore have broad implications for designing and implementing definitive notations.

Example 3: Observation of overlapping subsets

The tri-box framework abandons objective symbolic reference in preference for R- and W-sets of references to individual locations. This conceptual step allows highly specific references to subsets of state to be constructed that different agents can use in a more 'subjective' manner, as shown in Example 3. However, if reference to subsets is implemented in its full generality, the implications are extremely costly.

If in implementation we wish an R-set to be able to refer to any possible subset of the state S (i.e. any possible element of the powerset $\mathcal{P}(S)$), then the simplest fixed-length representation is a bit-vector, with one bit per value location representing presence or absence of that location in the reference. Unfortunately, such a representation requires as many bits as there are locations: using a 32-bit word for the representation only allows 32 locations. This problem is only partially redressed by the fact that, in the description of the framework above, we have not prescribed the length of the representation of a tri-box (although it is convenient to think of it conceptually as three boxes, it need not be represented in implementation as three value boxes).

However, in practice, it is unlikely that we will need to refer to every possible subset of S (i.e. every possible element of $\mathcal{P}(S)$). Many smaller (but less general) representations then become possible.

The DAM machine design implements a reference to a single location using a source pointer: a bit-vector interpreted as an address value. The source pointer is treated in the implementation as 'opaque' — it can only be used for dereferencing the particular location to which it points. Comparisons of source pointers or arithmetic on source pointers have no meaning — using terminology from the Java language, a DAM machine source pointer is a 'reference' rather than a 'pointer'. Allderidge's symbol table extension to the DAM machine in !Donald (see §3.3) persists in this design, as is conventional. Such a symbol table associates character string identifiers with values, but in the implementation, no significance is attached to structural relationships between identifiers, such as might stem (for example) from lexicographic ordering or spreadsheet cell naming conventions. It is therefore not usually meaningful to compare or perform arithmetic on character string identifiers.

The 'opaque-ness' of both of these forms of reference makes it necessary, when making a reference to multiple locations, to establish these by constructing lists of basic references (pointers or string identifiers).

A more general form of reference is provided in spreadsheet programs. For example, the "A1 reference style" in the spreadsheet program Excel (version 'X' for the Apple Macintosh) takes the following form (a regular expression synthesised from the documentation [exc01]):

$$\{\ SN\ \{\ \texttt{:}\ SN\ \}\ \texttt{!}\ \}\ \{\ CL\ \}\ \{\ RN\ \}\ \{\ \texttt{:}\ \{\ CL\ \}\ \{\ RN\ \}\ \}$$

where *SN* is a sheet name, *CL* is a column letter, *RN* is a row number, '`!`' is the exclamation character, '`:`' is the colon character, and curly brackets denote optional parts of the reference.

The colon delimits parts of the reference, some parts being optional. If *CLRN* is specified, then the reference denotes a single spreadsheet cell (e.g. `B3`). If all parts are specified, then each part of the reference specification can be thought of as a plane in three-dimensional space, the dimensions being columns, rows and 'sheets' of the current 'workbook' document. This reference representation (which has a fixed and small maximum length) therefore allows a reference to be made to any three-dimensional cube of cells (e.g. `Sheet1:Sheet3!A5:C7`).

Note however that it is not possible in the above syntax to reference a non-contiguous range of cells directly: although the formula `=SUM(A1:B2,D1:E2)` denotes the sum of the cells enclosed in the area `A1:E2` but not `C1:C2` (and therefore not a contiguous range), this formula is actually an invocation of the `SUM()` function with *two* reference arguments separated by a comma.

The tri-box framework assumes the most general form of reference possible, given a state S considered as a sequence of atomic 'boxes'. The provision of less general, but still powerful, forms of reference appears to involve two ingredients: organising state 'boxes' into some kind of space, and having a means to form references into that space. Ascribing a suitable structure to the space greatly increases the power of references used in combination. The 'suitability' of the structure will depend upon the domain[18].

Example 4: The `if` operator

The notion of D-in-S for higher-order dependency, made clear in the tri-box framework, is a conceptual breakthrough. Previously we have not been able to describe higher-order dependency in such concrete terms (see for example the discussion by Gehring *et al* in [GYC$^+$96]). An ill-defined abstract research problem is thus transformed into a precise technical problem. The technical problems remaining to be solved include the following three issues:

- The tri-box script graph data must be accessed indivisibly by concurrent agents, as are the value boxes. Note that the SR code in Appendix §5.A (p.328) does not implement D-in-S: the semaphores in the code mediate access to value boxes only.

[18] *Cf.* the way in which spreadsheets are suited to financial and administrative applications where tabular data is common.

- Higher-order dependency may complicate the detection of script graph cycles.
- Levels of evaluation (or change propagation) may need to be implemented. For maximum efficiency in Figure 5.15 (p.318) for example, if `b` and `x` are simultaneously changed, the `WRITEREF` operation must take place before the `COPY` operation.

Example 5: Subjective dependency

P-H. Sun first experimented with what is here termed 'subjective dependency' in his `dtkeden` extension to `tkeden` [Sun99] (see also §4.1.6), where `dtkeden` clients interact over a network via a `dtkeden` server, communicating by sending redefinition strings through TCP/IP sockets. Clients may have private local state and can maintain their own private dependencies between public data.

The tri-box framework lends clarity to what it means for two agents to have observables in common but to perceive different dependencies amongst them. In Figure 5.16 (p.320), each of the agents O1 and O2 has a different D. In this example, the two Ds are disjoint, but examples with some commonality between agent Ds are easily envisaged.

Implementing subjective dependency poses problems of distinguishing tri-boxes from value boxes and associating tri-boxes with agents. The association is particularly (perhaps only) important when synchronising agent action and observation, so one implementation design would be for an agent to identify its particular D when a protocol is used. The tri-box conceptual framework seems general enough to describe implementations ranging from distributed (with the state S distributed amongst multiple computers and the protocols implemented using message passing) to shared-memory (with the state S existing in the shared memory and protocols implemented using atomic processor instructions) to single-processor (with the state S existing in the single memory and a scheduler determining which agent to animate next on the basis of information resulting from protocol calls).

The diversity of the implementation issues raised by the above five examples of applying the tri-box framework suggests that no single ideal general purpose definition maintainer implementation exists. There may be an ideal if we restrict our usage and expectations of the tool to the types of Empirical Modelling performed with `tkeden` before 1999. However, new developments in applying EM, such as adding 3D graphical realisation and real-time input, lead directly to problems of organising the state space and concurrent real-time issues to which the best solution will depend both upon the requirement and the hardware available.

5.A Concurrent definition maintenance in SR

```
# A concurrent agent-based definition maintainer case study
# Ashley Ward (ashley@dcs.warwick.ac.uk)
# June-July 2003

resource agentdep()

  type debugprint = string[6]

  # upper bound on dependency table index.  Table is indexed 1:ND
  const ND := 5

  # upper bound on value store.  Store is indexed 1:NS
  const NS := 9

  type dref = int # 1:ND
  type sref = int # 1:NS

  type drefset = [ND] bool
  type srefset = [NS] bool
  type refset = [*] bool

  const EMPTYDREFSET := ([ND] false)
  const EMPTYSREFSET := ([NS] false)
  const FULLDREFSET := ([ND] true)

  # the store of values
  var s[NS]: int

  # the dependency table.  The order of arguments is sometimes
  # important (eg subtraction), but the table does not encode
  # each dependency precisely, just the info needed to describe the
  # dependency tree.  The exact implementation of each dependency is
  # described within the relevant u.

  var o[ND]: srefset := ([ND] EMPTYSREFSET)
  var i[ND]: srefset := ([ND] EMPTYSREFSET)

  type opid = enum(ADD, TIMES, MAX3, POWER)
  var u[ND]: opid
  op update(dref; debugprint) {send, call}

  sem l[ND] := ([ND] 1)

  var undef[ND]: bool := FULLDREFSET

  # end of dependency table

  type lockopt = enum(LOCK, UNLOCK)
  type recurseopt = enum(RECURSE, NORECURSE)

  /*
   * refset functions
   */

/*
  procedure nill()
  end
*/

  # bit-wise OR two drefsets together
  procedure ordrefset(a: drefset; b: drefset) returns aorb: drefset
    var i: int

    aorb := EMPTYDREFSET

/*
    co (i := 1 to ND st a[i] or b[i]) nill() -> aorb[i] := true oc
*/

    fa i := 1 to ND ->
      if a[i] or b[i] -> aorb[i] := true; fi
    af

  end

  # bit-wise OR two srefsets together
  procedure orsrefset(a: srefset; b: srefset) returns aorb: srefset
    var i: int

    aorb := EMPTYSREFSET

    fa i := 1 to NS ->
      if a[i] or b[i] -> aorb[i] := true; fi
    af

  end

  # bit-wise AND two srefsets together
  procedure andsrefset(a: srefset; b: srefset) returns aandb: srefset
    var i: int

    aandb := EMPTYSREFSET

    fa i := 1 to NS ->
      if a[i] and b[i] -> aandb[i] := true; fi
    af

  end

```

```
# is the given refset empty (ie all false)
# or does it contain a true value?
procedure isemptyrefset(r: refset) returns empty: bool
  var i: int

  empty := true

  fa i := 1 to ub(r) ->
    if r[i] ->
      empty := false
      exit
    fi
  af

end

# convert a refset to a string for debug printing purposes
procedure refsettostring(r: refset) returns s: string[80]
  var i: int

  s := ""

  fa i := 1 to ub(r) ->
    if r[i] ->
      s := s || string(i)
    fi
  af

end
```

```
/*
 * procedures to manipulate the dependency table
 */

# recursively all sources (D space) of all r (S space), including r
procedure dsources(r: srefset) returns ret: drefset
  var d: dref

  ret := EMPTYDREFSET

  fa d := 1 to ND ->
    if not isemptyrefset(andsrefset(o[d], r)) ->
      ret[d] := true
      ret := ordrefset(ret, dsources(i[d]))
    fi
  af

end

# recursively? all targets (D space) of all r (S space), excluding r
procedure dtargets(r: srefset;
                   recurse: recurseopt) returns ret: drefset
  var d: dref
  var ts: srefset

  ret := EMPTYDREFSET

  fa d := 1 to ND ->
    if not isemptyrefset(andsrefset(i[d], r)) ->
      ret[d] := true
  if recurse = RECURSE ->
    ret := ordrefset(ret, dtargets(o[d], recurse))
  fi
    fi
  af

end
```

```
# lock/unlock this set of Ds
procedure lockset(lock: lockopt; locks: drefset; dp: debugprint)
  var d: dref

  if lock = LOCK ->
    # as long as concurrent competing processes make their locks in the
    # same sequential order, deadlocks are prevented

    fa d := 1 to ND ->
      if locks[d] ->
        write(dp, "P(", d, ")")
        P(l[d])
      fi
    af

  [] lock = UNLOCK ->
    /* co (d := 1 to ND st locks[d]) V(l[d]) oc */

    fa d := 1 to ND ->
      if locks[d] ->
        write(dp, "V(", d, ")")
        V(l[d])
      fi
    af

  fi

end

# are any of these s marked as undefined in the dependency table?
procedure containsundef(sources: srefset) returns containsundef: bool
  var sr: sref
  var dr: dref

  containsundef := false

  fa dr := 1 to ND ->
    if not isemptyrefset(andsrefset(o[dr], sources)) ->
      if undef[dr] ->
        containsundef := true
        exit
      fi
    fi
  af

end

```

```
/*
 * observe / change protocol procedures
 */

# prohibit change (& observation?) of all of r and sources
# so that state appears consistent
procedure preOlock(r: srefset; dp: debugprint) returns locked: drefset
  # lock any use of any r and all sources down the tree
  locked := dsources(r)

  lockset(LOCK, locked, dp)

end

# allow change of previously locked dependencies
op postOunlock(drefset; debugprint) {send}
proc postOunlock(locked, dp)
  lockset(UNLOCK, locked, dp)

end

# prohibit observation (& change?) of all of r and targets
procedure preClock(r: srefset; dp: debugprint) returns toupdate: drefset
  var targets: drefset

  # lock any targets of all r and all targets up the tree (excluding r)
  targets := dtargets(r, RECURSE)

  lockset(LOCK, targets, dp)

  # now mark all targets (excluding r) up the tree as undefined
  undef := ordrefset(undef, targets)

  # the first level of targets of r should now be updated
  # (and recursively, the targets of those targets)
  toupdate := dtargets(r, NORECURSE)

end

# update first level of dependencies, allow observation/change, then
# propagate dependency update up to the next level
op postCupdateunlock(drefset; debugprint) {send, call}
proc postCupdateunlock(toupdate, dp)
  var d: dref

  # in parallel, invoke the necessary update procs, which should each
  # read their input, write output, unlock, then invoke the next level
  # of dependency update
  co (d := 1 to ND st toupdate[d]) call update(d, dp) oc

end
```

```

/*
 * update dependency procedures
 */

# dirty hack for getting the refno'th single reference out of a set
procedure oneref(set: srefset; refno: int) returns s: sref
  var i: sref

  s := 0

  fa i := 1 to NS ->
    if set[i] and (--refno = 0) ->
      s := i
      exit
    fi
  af

end

# update the s outputs of a dependency.  This is called from update and
# also possibly manually when a dependency is changed
procedure valueupdate(d: dref)
  if u[d] = ADD ->
    s[oneref(o[d], 1)] := s[oneref(i[d], 1)] + s[oneref(i[d], 2)]
  [] u[d] = TIMES ->
    s[oneref(o[d], 1)] := s[oneref(i[d], 1)] * s[oneref(i[d], 2)]
  [] u[d] = MAX3 ->
    s[oneref(o[d], 1)] := max(s[oneref(i[d], 1)],
                              s[oneref(i[d], 2)],
                              s[oneref(i[d], 3)])
  [] u[d] = POWER ->
    s[oneref(o[d], 1)] := s[oneref(i[d], 1)] ** s[oneref(i[d], 2)]
  fi

end
```

```
# update state for a particular dependency by reading input and writing
# output, then unlock the dependency and invoke the next level of
# dependency update
proc update(d, dp)
  var args: int
  var sr: sref

  write(dp, "UPDATE", d)

  if containsundef(i[d]) ->
    # at least one source value is undefined: ignore now,
    # don’t propagatechange upwards and wait for the final update call
    write(dp, "UPDATE undefined source")
    return
  fi

  if not undef[d] ->
    # this d has already been updated
    write(dp, "UPDATE value already defined")
    return
  fi

  valueupdate(d)

  # the output value is no longer undefined
  undef[d] := false

  write(dp, "V(", d, ")")
  V(l[d])

  call postCupdateunlock(dtargets(o[d], NORECURSE), dp)

end


```

```
/*
 * debug test code
 */

procedure writesd()
  var sr: sref
  var dri: dref, dr: dref

  fa sr := 1 to NS ->

    # find this s in D space
    dr := 0
    fa dri := 1 to ND ->
      if o[dri][sr] ->
        dr := dri
        exit
      fi
    af

    writes(" S:", sr, " = ", s[sr])

    if dr != 0 ->
      writes("\tD:", dr,
             " o:", refsettostring(o[dr]),
             " i:", refsettostring(i[dr]),
             " u:", u[dr],
             " l:?", # can't print l[dr]
             " undef:", undef[dr],
             "\n")
    [] else ->
      write()
    fi

  af

end
```

```
# "Power script" - example from Cartwright p. 123

o[1] := ([6] false, true, [2] false); # g
i[1] := ([4] false, true, true, [3] false); # e, f
u[1] := TIMES;

o[2] := ([8] false, true); # i
i[2] := (true, [6] false, true, false); # a, h
u[2] := POWER;

o[3] := ([7] false, true, false); # h
i[3] := ([3] false, true, false, true, true, [2] false); # g, f, d
u[3] := MAX3;

o[4] := ([4] false, true, [4] false); # e
i[4] := (true, true, [7] false); # a, b
u[4] := ADD;

o[5] := ([5] false, true, [3] false); # f
i[5] := (false, true, true, [6] false); # b, c
u[5] := ADD;

# initialise values but not dependencies
s := (1, 2, 3, 4, [5] -1)
undef := FULLDREFSET

writesd()

# simulate change to all non-d to initialise.  starting with just a
# (1) or c (3) won't work as then e (4) or f (5) will be undefined.
var sr: srefset
sr := ([4] true, [5] false) # a,b,c,d

var toupdate: drefset
toupdate := preClock(sr, "INIT")

write("TOUPDATE", refsettostring(toupdate))

postCupdateunlock(toupdate, "INIT")

writesd()

# observe g
sr := EMPTYSREFSET
sr[7] := true # g
var locked: drefset
locked := preOlock(sr, "OG")
write("LOCKED", refsettostring(locked))
write("S7=", s[7])
send postOunlock(locked, "OG")

```

```
/*
 * Concurrent test processes
 */

# observe a single s value.  Perhaps not useful as no possibility for
# simultaneity of observation.
procedure Os(sr: sref; dp: debugprint) returns v: int
  var sset: srefset
  var locked: drefset

  sset := EMPTYSREFSET
  sset[sr] := true

  locked := preOlock(sset, dp)
  v := s[sr]
  send postOunlock(locked, dp)

end

# change a single s value (not a d)
procedure Cs(sr: sref; v: int; dp: debugprint)
  var sset: srefset
  var toupdate: drefset

  sset := EMPTYSREFSET
  sset[sr] := true

  toupdate := preClock(sset, dp)
  s[sr] := v
  send postCupdateunlock(toupdate, dp)

end
```

```
/*
  # simple observation of just one value at a time, observing dependency
  # constraint (although if there is no simultaneity, this is
  # questionable)
  process O1
    var i: int

    fa i := 1 to 20 ->
      nap(int(random(100)))
      write("O1 S7", Os(7))

    af
  end

  # simple observation of just one value at a time
  process O2
    var i: int

    fa i := 1 to 20 ->
      nap(int(random(100)))
      write("O2 S7", s[7])

    af
  end
*/

```

```
# Od observes abcefg simultaneously and perceives the dependencies
process Od
  var i: int
  var sset: srefset
  var locked: drefset
  var dp: debugprint := "   Od"

  fa i := 1 to 20 ->
    nap(int(random(100)))

    sset := ([3] true, false, [3] true, [2] false) # abcefg

    locked := preOlock(sset, dp)
    writes(dp, ": S1=", s[1], " S2=", s[2], " S3=", s[3],
           " S5(1+?*2)=", s[5], "(", s[5]=s[1]+s[2], ")",
           " S6(2+3)=", s[6], "(", s[6]=s[2]+s[3], ")",
           " S7(5*6)=", s[7], "(",s[7]=s[5]*s[6], ")\n")
    send postOunlock(locked, dp)

  af

end

# On observes abcefg simultaneously, but does not perceive dependency
process On
  var i: int
  var dp: debugprint := "    On"

  fa i := 1 to 20 ->
    nap(int(random(100)))

    writes(dp, ": S1=", s[1], " S2=", s[2], " S3=", s[3],
           " S5(1+?*2)=", s[5], "(", s[5]=s[1]+s[2], ")",
           " S6(2+3)=", s[6], "(", s[6]=s[2]+s[3], ")",
           " S7(5*6)=", s[7], "(",s[7]=s[5]*s[6], ")\n")

  af

end
```

```
# C1 changes the value of S1 (a), observing dependency action
# constraints
process C1
  var i: int
  var randomv: int
  var dp: debugprint := "C1"

  fa i := 1 to 20 ->
    nap(int(random(100)))

    randomv := int(random(10))
    write("C1: S1=", randomv, "...")
    Cs(1, randomv, dp)
  af

end

# C2 changes the value of S2 (b), observing dependency action
# constraints
process C2
  var i: int
  var randomv: int
  var dp: debugprint := " C2"

  fa i := 1 to 20 ->
    nap(int(random(100)))

    randomv := int(random(10))
    write("C2: S2=", randomv, "...")
    Cs(2, randomv, dp)
  af

end
```

```
  # C3 changes D4 (value at S5) between TIMES and ADD, observing
  # dependency action constraints
  process C3
    var i: int
    var sset: srefset
    var toupdate: drefset
    var dp: debugprint := "  C3"

    # whether D4 is TIMES or ADD (note operators chosen such that errors
    # cannot occur with source values of 0)
    var t: bool := true

    fa i := 1 to 20 ->
      nap(int(random(100)))

      sset := EMPTYSREFSET
      sset[5] := true # e

      toupdate := preClock(sset, dp)

      writes("C3: S5 becomes ")
      if t ->
        write("TIMES")
        u[4] := TIMES  # note dref, not sref
      [] else ->
        write("ADD")
        u[4] := ADD
      fi

      # have to recalculate the s value manually for a d change
      write(dp, "VALUEUPDATE 4")
      valueupdate(4)

      send postCupdateunlock(toupdate, dp)

      t := not t

    af
  end

end
```