3 Abstract Definitive Modelling

This chapter introduces the Abstract Definitive Modelling (ADM) framework for MWDS in all its aspects. The versatile use of definitive scripts in representing a variety of agent perspectives is illustrated by discussing how a wide range of models can be construed as ADM artefacts. These indicate the capacity for the ADM framework to support 'universal agent-oriented modelling'.

3.1 The Abstract Definitive Modelling framework

In this section, the ADM framework is motivated with reference to the wide diversity of ways of using MWDS. Throughout the thesis so far we have shown that MWDS can be applied to both:

- open-development and closed-world modelling (cf. Figures 1-5 and 1-6) – contrast the exploratory Number model (cf. Figure 2-16) with the program-like Jugs model subject to the preconceived pattern of interaction supported by its graphical interface (cf. Figure 2-5);
- single-agent and multi-agent scenarios (cf. Figures 2-18 and 2-19) – contrast the architect's single-agent model of the door (cf. Figure 2-6) with the user and animating agents of the Jugs model (cf. Figure 2-5);
- internal and external semantic relations (cf. Figures 2-12(a) and (b)) – contrast the Triangle model (cf. Figure 2-14) with the Number model (cf. Figure 2-16).

Within a multi-agent scenario, MWDS can also be used to represent:

- construal and specification (cf. Figure 2-17);
- external and internal agents (cf. Figures 2-20(a) and (b));
- distributed and non-distributed models.

The above review shows that MWDS is very broad and embraces both human-centred and automatic activities. This breadth stems from the fact that the human interpreter is an integral part of MWDS – as Figure 2-11 illustrates, a definitive script is not attributed an abstract meaning in isolation from an observer and a situation. In developing the Jugs model, the modeller may be concerned with construing possible operations and interactions involved in pouring liquid between real-world jugs, or (as was the case in the original development [Bey89a]) with replicating the behaviour of a procedural program. The modeller may need to project him/herself into different agent roles, including external agents such as the user of the Jugs model (teacher or pupil) and internal agents such as the init_pour and pour mechanisms. The purpose and the nature of such projection frame the perspective of the modeller, the mode of observation and the character of observables.

In MWDS, there is an essential role for three capacities of the human mind in:

- the 'comprehension' of state;
- the perception of continuity through change (cf. William James's concept of 'continuous relation' [James96]);
- the reinterpretation of the 'same' experience through its transposition to a new context (cf. Mark Turner's concept of 'blending' [Turn96]).

With reference to Figures 1-5 and 1-6, these activities correspond respectively to three problematic issues in closed-world modelling: combining two closed-world models into one; referring to a state within the original closed-world model in the new revised closed-world model; and revising the semantic relation without changing the model.

The three capacities mentioned above will be exemplified and elaborated with reference to an illustrative exercise in MWDS. The first step in the modelling exercise is to bring together the two scripts: the Jugs and Door model as depicted in Figure 2-5 and Figure 2-6 respectively. This involves a comprehension of state whereby the set of definitions in the two scripts are viewed as a single definitive script to be interpreted in relation to a new situation (cf. Figure 2-11). This reflects a commonplace human activity of bringing together disparate or spatially distributed observables in the mind so as to form what is conceptually a single artefact. There is also continuity in the modeller's conception of the artefact; he/she regards the Jugs-and-Door artefact as the Jugs artefact (cf. Figure 3-1) but having a different character corresponding to a change in situation. The modeller makes the association between the Jugs-and-Door artefact and the Jugs

artefact rather than between the Jugs-and-Door artefact and the Door artefact[1] because he/she sees the potential for reinterpreting a jug as a water tank and a door as a valve within the tank. This potential is realised in MWDS by making two copies of the Door script to represent a valve for each tank and pasting the corresponding line drawings over the bottoms of the jugs. The final step in the modelling exercise is to establish dependencies between the open/closed status of the valves and the validity of menu options. This involves the invention of a suitable construal, such as: *You cannot fill tank A or pour from tank B to tank A when valveA is open* and vice versa.

The above agenda applies recursively via projection onto other agents' roles. Such projection is the essential principle in construing agency associated with the situated model. There are many ways to construe the situation and this leads to an enormous variety of different ways of viewing agents and agency.

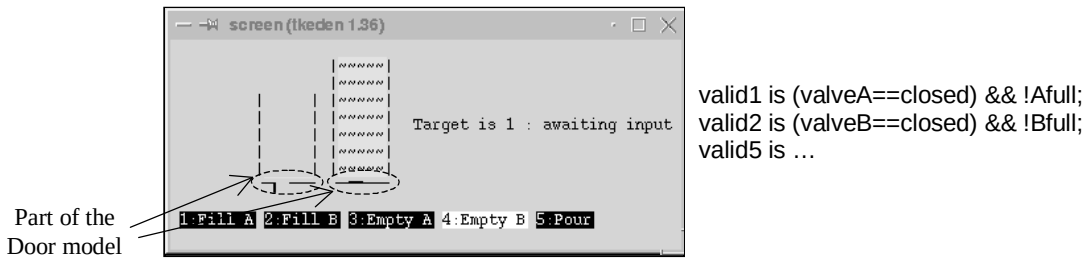


Figure 3-1: The Jugs-and-Door artefact

Agents as active instigators of change: In construal, a primary activity is attributing state-change to agents. For instance, when the valve opens we will expect the tank to empty. This can be interpreted as an event ('valve A opens') such that the jug empties. This can be expressed using an LSD protocol:

(valveA_opens) && (contentA>0) → contentA=contentA-1;

In this interpretation, the context for observation is such that a "valve-opening event" is meaningful, and the valve-opening action is attributed to valve A as an agent. The modeller's likely purpose is explanation ('tank A empties because valve A opens'). An alternative way of observing the situation might correspond to an expectation ('tank A empties when valve A is open'), rather than an explanation, that can be conveniently expressed using an LSD protocol:

(valveA==open) && (contentA>0) → contentA=contentA-1;

Whereas the event valveA_opens presumes observation of motion, the observation valveA==open can be performed statically.

[1] Associating the Jugs-and-Door artefact with the Door artefact might be interpreted as 'making a cupboard'.

Automatic agents and circumscribed change: The LSD account in Figure 3-2 is the counterpart of the tkeden implementation of the Jugs program depicted in Figure 2-5. It documents the state-changing protocols of agents that the modeller devises for manipulating the jugs in order to specify the init_pour and pour actions in the tkeden implementation. The modeller develops this account of the necessary agents by projecting him/herself on each agent's role. In this case, the LSD account can also be viewed as a program specification (cf. Slade [Slade89]).

```
agent jug-user{
  handle   (int) capA,
           (int) capB,
           (int) input
  protocol
   valid1 → input=1;
   valid2 …
}
```

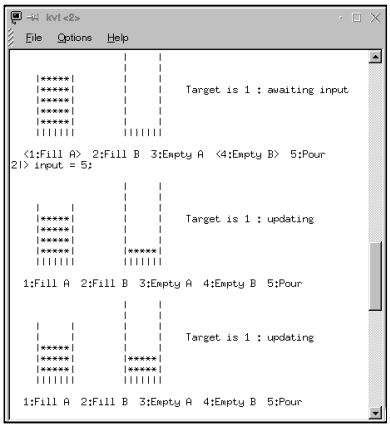
```
agent pour{
  handle   (int) contentA,
           (int) contentB,
           (int) step,
           (int) updating
  oracle   (int) option,
           (int) step
  protocol
    (option==1)&&(valid1) →
                  contentA=contentA+1;
                   pour();
    option==2 …
}
```

```
agent init_pour{
  handle   (int) step,
           (int) option,
           (int) updating
  oracle   (int) input,
           (int) step
  protocol
   input==1 → updating=1;
              option=1;
              pour();
   input==2 …
}
```

Figure 3-2: An LSD account/specification of the Jugs model/program

Passive agents and latent change: In an LSD account, we can have LSD agents without any privileges. The grouping of observables into the states of a single agent still has a purpose, for example in identifying objects as groups of observables that come in to and go out of existence together. Agents without protocols can be viewed as passive agents, but they may still play the key role in maintaining state (cf. the Battery and the Light in Figure 2-20). In the course of model development, an agent may migrate from passive to active. For instance, the bottom can fall out of a jug when the level of liquid is too high.

The computer as an agent: In MWDS, the modeller is centrally concerned with the role of the computer as a state-changing agent. The emphasis is on how the internal semantic relation can be crafted by using appropriate definitive notations and the external state-generating capabilities of the machine. This use of the computer as an instrument [BCH+01] is as much art as science. By way of illustration, pasting the window that contains the Door model onto the Jugs model to form the artefact depicted in Figure 3-1 exploits an incidental feature of the design of Scout windows in an opportunistic way. The kind of definitive script used may also reflect the different capabilities of notations and devices available to the modeller (cf. the text interface for the jugs on ttyeden, as depicted in Figure 3-3).




```
_display is displayabove(jjt, menuform);
jjt is displayrt(jugA, displayrt(jugB,displayrt(targt,statt)));
jugA is jugdisplay(height,capA,widthA,contentA);
func jugdisplay{
   auto s,i;
   for( i; ; ){ append s, jugline(i-1,…)} return t;
}
func jugline{
   auto t, r, s, c;
   r = repchar(‘ ’, $3-1); …
   t = r // c // s // c // r;
   return t;
};
func repchar{};
func displayabove{};
…
```

Figure 3-3: Screenshot and script extract for the text-interface Jugs model

The modeller as an agent: The role of the modeller is very rich in MWDS since he/she can take various forms of an agent during modelling such as the role of:

- an external observer or interpreter. The modeller comprehends what is going on in the artefact with reference to his/her observation and perception. These can be influenced by many factors such as his/her physical ability (e.g. whether they are colour-blind) and his/her personal experience, expectation, and knowledge of interactions in the situation. All these issues contribute to effective and repeatable experiments. They can also lead to the identification of reliable patterns of interaction between agents. As the models discussed above illustrate, the external observer’s construal may include elements of open exploratory interaction and closed program-like behaviour.
- an actor or participant. In this role, the modeller construes a multi-agent situation from the perspective of an internal agent. If the modeller is an external observer of (respectively, a participant in) the situation, this demands an ability to project him/herself as an internal agent (respectively, an external observer) to view and see things from the point of view of that agent. Such projection involves issues such as his/her powers of imagination, speed of response, and skill in observation. In open development, the modeller can assume the role of a super-agent in which he/she can arbitrarily change and redesign other agents’ roles and can also act as a meta-level agent who resolves conflict and synchronise actions.
- a director. In this role, the modeller directs the actions of internal agents in executing a specific process. This activity is like directing a play or the manual execution of a concurrent program. The modeller has the freedom to intervene and interrupt the running model. This is very useful when we want to understand, debug and experiment with processes in the model.

The discussion so far shows that MWDS is very broad and flexible and it is useful to have a framework in which to consider the above agenda in a coherent way.

The Abstract Definitive Modelling (ADM) framework

We shall describe an abstract framework for MWDS in all its aspects by revisiting the 'abstract definitive machine' (adm). The adm concept was first developed in connection with the possible use of LSD for concurrent systems specification and the semantics of Eden programming. It was first proposed as an abstract machine model for conventional applications (viz. specifying the behaviour of concurrent systems and 'definitive programming') of MWDS. In the process of investigating these applications, the adm emerged as 'more than a machine' and 'not of its essence machine-like' [Slade90]. (In MWDS, if we try to give a closed-world semantics, we limit our perception of the external world and lose the essential access to open-ended experiment.) We will use the adm here as a setting in which to frame all the diverse uses of MWDS. When the adm is used in this way, it is no longer appropriate to regard it as an abstract *machine*. In this context, we shall instead adapt the acronym 'adm' and adopt the acronym 'ADM' to refer to the 'Abstract Definitive Modelling' framework. The Abstract Definitive Modelling framework is a very broad concept that can be viewed from both a machine perspective (when it corresponds to a machine architecture based on MWDS) and a human perspective (when it corresponds to a conceptual framework for multi-agent construal).

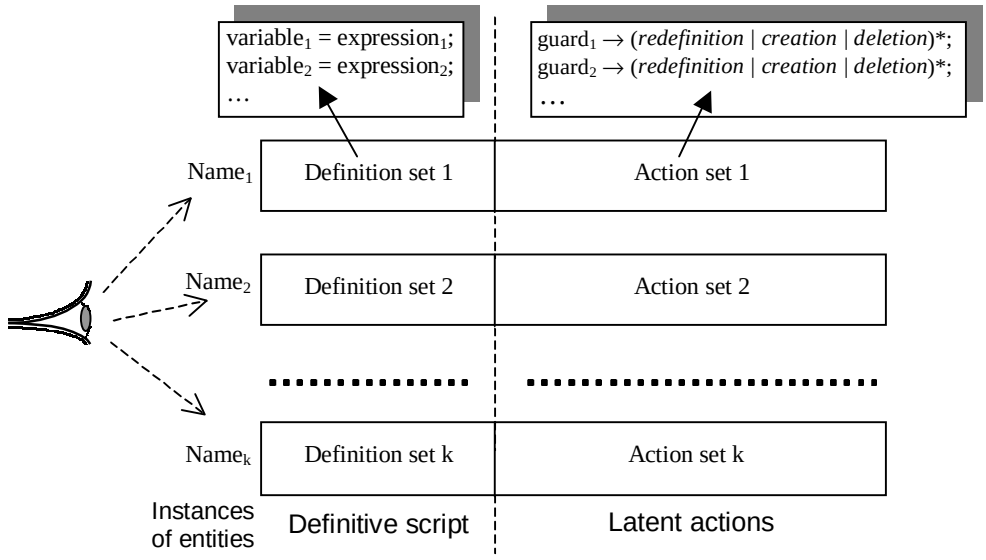


Figure 3-4: The ADM from the machine perspective

An ADM artefact consists of instances of entities. Each entity comprises a set of definitions and a set of actions and each instance of that entity has an identifying name. Each

action is a guarded sequence of redefinitions, entity invocations or deletions (cf. Figure 3-4). The ADM artefact is only meaningful in relation to a human interpreter. The primary role of the interpreter is in bringing together the set of definitions associated with entity instances in the artefact so that they are being viewed as a single definitive script (cf. the discussion of comprehension of state above).

The ADM artefact is a representation of a pure definitive model that is richer than an un-annotated definitive script. There is no direct translation from an LSD account to an ADM artefact, but there is a loose correspondence between agents and entities, protocols and actions, and derivates and definitions. With reference to Figure 2-11, the ADM enables us not merely to represent features of an artefact in isolation but also in the context as established by the modeller's implicit knowledge and orientation towards the situation. In particular, an ADM artefact can represent a current state with reference to what the observer associates with the state of, and interaction with, the definitive script. How we observe the referent and how we interpret actions in the external world depends on the context and purpose of the modelling. In construing situations we are concerned with attributing change to agent actions. In specifying a system we are concerned with prescribing agent actions.

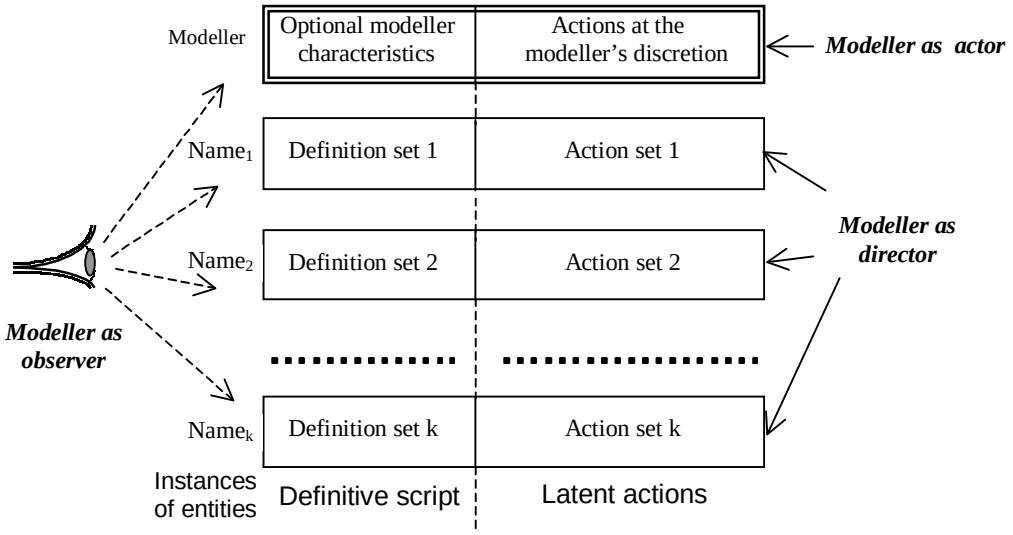


Figure 3-5: The ADM from the human perspective

The simple view of the ADM depicted in Figure 3-4, where the modeller's role is restricted to that of an external observer, is associated with multi-agent scenarios where the concurrent interaction of the agents is essentially circumscribed. In such a scenario, the relationship between the modeller and the ADM artefact is similar to the relationship between the user and a traditional application. In interpreting the ADM as a machine (viz. the adm), we regard

the actions as guarded sequences of commands to be executed when guards are true in the definitive script. The adm was first developed as an abstract model for parallel computation based on the notion of concurrent redefinition depicted in Figure 2-1. This is described in [BSY89] where the use of the adm to represent a systolic array architecture is illustrated.

User interaction in the adm embraces activities such as the manual execution and debugging of a program – the user can act as a meta-agent to direct agents in synchronising actions and resolving conflicts. A detailed account of the role of the adm in giving an operational interpretation to an LSD specification or to an Eden program is given in [Slade90]. Further discussion of the adm is beyond the scope of this thesis.

In general, the modeller's interaction within the ADM is much broader than the adm supports. The progression of state is not normally automatic, but can proceed in an open-ended manner under human control. All the roles of the modeller as an agent discussed above can be interpreted within the ADM. This is illustrated in Figure 3-5 where the modeller is represented by an additional entity in which relevant characteristics and state observables of the modeller are represented by definitions and open-ended interactions at the discretion of the modeller are represented by actions.

An illustrative example of an ADM artefact

The ADM artefact depicted in Figure 3-6 illustrates how the ADM framework can be applied to the Jugs model (cf. Figures 2-5 and 3-3). The ADM artefact enriches the definitive model implemented in tkeden that is depicted in Figure 2-5 in several respects:

- by grouping definitions and actions into entities to represent a multi-agent construal of the Jugs model (cf. Figure 2-19 and Figure 3-2);
- by adding actions to represent the privileges of human agents;
- by framing the stimulus-response patterns for automatic agents as actions.

The definitions and actions in the artefact all refer to the computer-based states and transitions associated with the model. With reference to Figure 2-12, these states and transitions can typically be interpreted in connection with both the internal and external semantic relations. This interpretation of the definitions and actions allows us to informally classify entities according to their role in respect of these semantic relations, as elements of the interface, of the internal state-transition model and of the external entity-relationship model.

With reference to Figure 3-6, the internal agents considered serve as the interface and the automatic mechanism. The pour agent is only intermittently alive, when it is needed to act. An alternative interface, such as the text-based presentation of the jugs JugAtxt (cf. Figures 3-6 and 3-3), can be introduced into the artefact as a supplementary or replacement feature. The artefact also includes entities to represent the external referent (e.g. JugA).

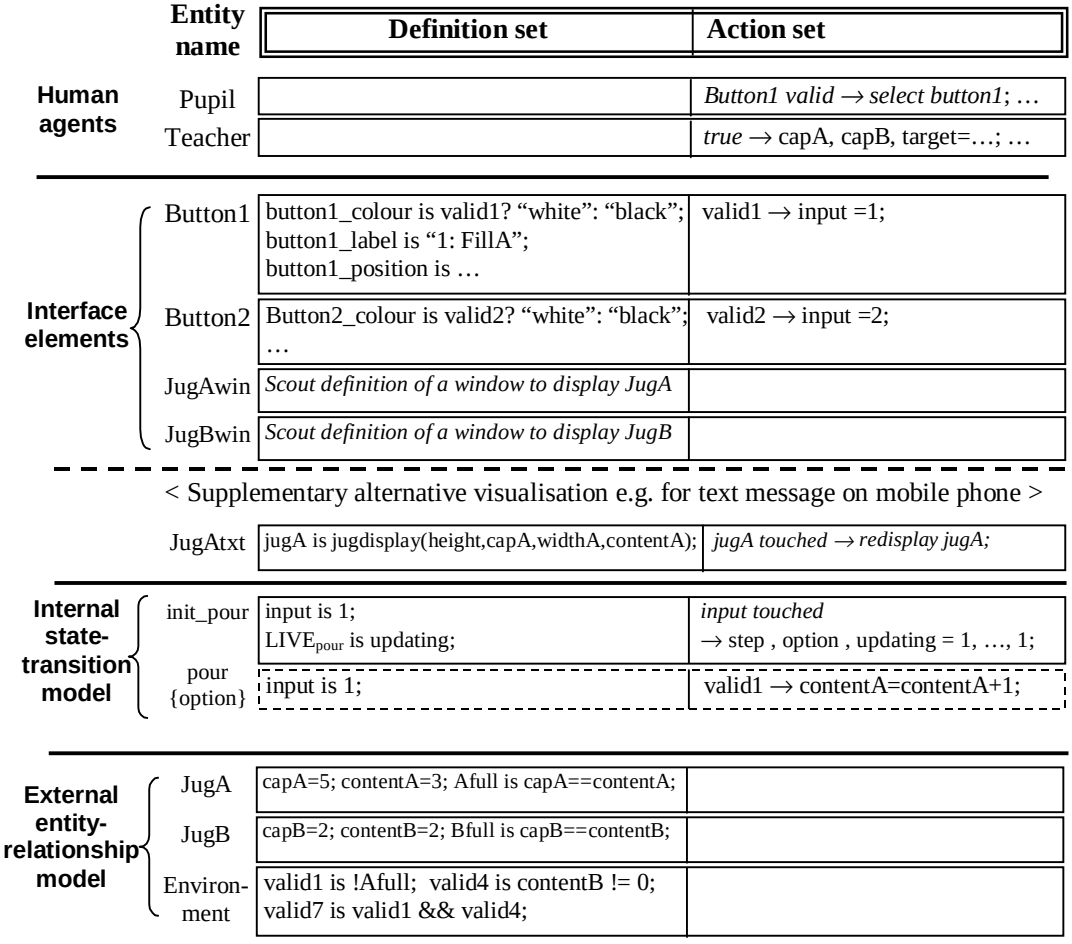


Figure 3-6: An ADM artefact for Jugs

The ADM artefact can be interpreted both as a specification of the Jugs program and (with more difficulty) as a construal of a situation in which a teacher and a pupil interact with real-world jugs. In the former (respectively, the latter) interpretation, the protocol for the pupil corresponds to a button selection (respectively, an action such as filling a jug) and the protocol for the teacher corresponds to parallel redefinitions of capA, capB and target (respectively, supplying a new pair of jugs and another problem scenario). Other scenarios for use of the Jugs artefact will be explored in Chapter 5.

The grouping of definitions into entities and the informal classification of entities in Figure 3-6 illustrate a general feature of applying definitive principles in the representation of state. Subsets of definitions can be extracted from a script and interpreted as representing a particular feature (e.g. Button1 in the Jugs interface) or broader aspect (e.g. the interface elements) of the entire state. Such a subset can be interpreted in isolation from the model, subject to the modeller providing the appropriate context (see for example the line drawing extracted from the Car History model in Section 3.2.3, as depicted in Figure 3-12). We shall refer to the aspects of state of the Jugs model identified in Figure 3-6 in the review of definitive models in the next section.

In Figure 3-6, the classification of JugA and JugB as referring to entities and relationships in the external referent is a matter of interpretation. On the one hand, the definition of JugA has more to do with an external observable than the Scout points that determine the position of the windows that display the jugs on the screen. On the other hand, JugA can also be viewed as an essential part of the internal model that determines the state of the screen. The two interpretations for JugA relate to the external and internal semantic relations respectively (cf. Figure 2-12) and they accordingly reflect the respective perspectives of a user and a programmer.

Giving a sharp characterisation of observables in the ADM artefact is problematic when we consider that the fundamental semantic mechanism operating in MWDS (cf. Figures 1-5 and 2-11) is matching of experiences. As the overlap between the internal and external semantic relation in Figure 2-12 shows, it is possible to blur the distinction between points and lines as declared and defined in the DoNaLD script and the corresponding points and lines on the computer screen (cf. Section 2.2.1). This issue relates to the difficulty of declaring a boundary for an ADM artefact: consider for instance, the way in which the bottom of the jug A becomes identified with the valve in the Jugs-and-Door variant of the Jugs artefact. It is important to note the contrast between the fluid classification of observables in Figure 3-6 and the more robust classification of observables to be introduced in Chapter 5. As will be further discussed in Chapter 5, this is a consequence of the 'artificial' boundary we impose upon an ADM artefact when viewing as a device, and the restrictions we put upon interaction with it in appropriate or near-appropriate use.

3.2 Illustrating the ADM framework

The discussion of definitive models in this section has two purposes. It illustrates the diverse and broad ways to apply MWDS in constructing definitive models with reference to the ADM framework proposed in the last section. It also gives more background on the practical use of (d)tkeden as a tool for building an ADM artefact and the technical issues this raises. The models in this section can be divided into four categories according to characteristics of their construction and the perspective of the modeller. They deal with:

- observation and interpretation in single-agent modelling;
- comprehension of state in multi-agent modelling;
- multi-agent modelling for the internal semantic relation;
- open and closed interaction.

The illustrative models supply practical examples of all aspects of agency that can be supported within the ADM framework (cf. Section 3.1). Roles of the modeller as an agent include an architect, teacher, mathematician, game designer and user interface designer. The models demonstrate different perspectives of the modeller as an external agent or a participant. They also reflect different purposes, such as exploratory modelling, instrument building and product design. Different perspectives and purposes can be represented in the modeller's interaction with a single model and the modeller's perception of the model can change in accordance with his/her evolving experience.

3.2.1 Observation and interpretation in single-agent modelling

In this section we will discuss how a definitive script can be used to represent an internal semantic relation in which the observables in the internal referents are geometric entities such as line, point and circle, and also how these can have very different interpretations in the external semantic relations (cf. Figure 2-12). The focus is on constructing geometric models based on a single-agent perspective, where the agents, in this case, are an architect and a mathematician. The virtues of using definitive scripts in representing geometric entities for an architect who wants to design a room (the Room Viewer model) and a mathematician who needs to explore his/her mathematical theory (the Lines model) will be discussed.

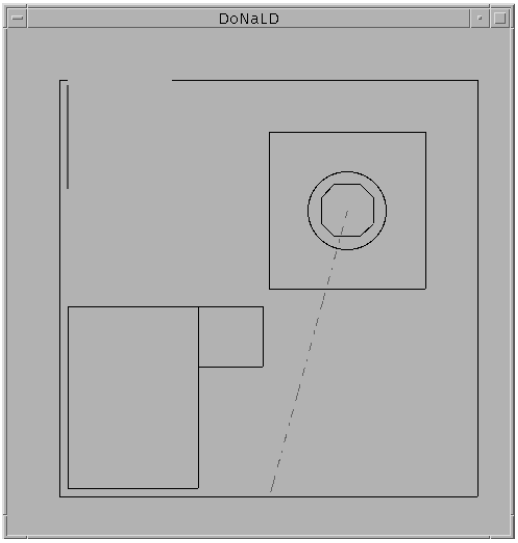
In both models, a DoNaLD script plays a crucial role in representing the inter-relationship and dependency between geometric entities. For instance, a line is defined with reference to its endpoints – relocating an endpoint will indivisibly affect the state of the corresponding line (cf.

Figures 2-6 and 2-14). A similar use of dependency in geometric modelling for other applications, such as Wyvill's interactive graphics [Wyv75] and L.E.G.O [FPR85, FP88], has already been discussed in Chapter 1.

The Room Viewer model – Definition-based geometric representation

The Room Viewer model {Room90} uses a DoNaLD script to represent an architect's viewpoint on designing a room. As depicted in Figure 3-7, the room consists of a door, a table with a lamp on it, a desk with a drawer and a cable connected to the lamp. DoNaLD has data types such as integer, point, line, circle, shape and label (cf. Section 2.1.2) to serve as a line-drawing tool for the architect to design the room. Line drawings are displayed on the DoNaLD default screen (bottom left {0,0}, top right {1000,1000}) in accordance with their definitions.

Definitive variables in the model create references that directly represent the architect's view on designing the room. These references are different from traditional procedural variables (meaningful only during the execution of a program) and declarative variables (statically defined, like mathematical variables, independent of program execution). Definitions are used to create associations between values of variables in the script that reflect the architect's interpretation of these variables and expectations about how they are linked in interaction. This means that, by interacting with the model we can identify the real world entity represented by particular variables. This contrasts with the way in which the external interpretation of procedural and declarative variables is typically established by convention.




```
%donald
point NW, NE, SW, SE          #  the 4 corners of room
line N1, N2, S, E, W          # the 3 walls of room
openshape door                # plus 2 walls and the door
within door { … }
openshape table               # the definition of the table
within table{
   point NW, NE, SW, SE, centre
   SW = centre-{width div 2,length div2}
   centre = (~/SE + ~/NW) div 1.5 …
}
openshape lamp                # the lamp is on the table
within lamp{ centre =  ~/table/centre …}
openshape desk                # a desk with a drawer
within desk{
point NW, NE, SW, SE, centre
line N, S, E, W
        SW = ~SW+{15,15} …
        openshape drawer   # a drawer inside the desk
        within drawer{ NW= ~/NE … }
} …
```

Figure 3-7: A screenshot of the Room Viewer model and its script

Features of the model

The modeller, who acts as a room designer, conceives the room as containing some furniture by using lines to represent the boundary of the room and geometric displays to represent the furniture in the room. Defining lines as dependent on their pair of endpoints can help in comprehending scripts since in some contexts constructing lines in this way is consistent with the way people conceive the structure of the line.

The model evolves gradually through an architect or the user or the designer's interaction with it. They can adjust and refine the design of the room on-the-fly by redefining definitive variables either by changing their values or their definitions. Many aspects of what they observe in a real-world situation can be represented directly in the definitive script. By interpreting the script in Figure 3-7, we see that the designer can use the DoNaLD openshape feature to group the furniture, for instance the desk with its drawer. The openshape door is similar to the Door model discussed in Section 2.1.3. The architect defines the position of the furniture with reference to the four corners of the room represented by SE, NE, NW and SW as shown in the script in Figure 3-7. For instance, the position of the table is defined in terms of the ~/SE and ~/NW variables, which are the South-East and North-West corners of the room.

By referring to observable names and tracing the dependencies in the definitions, the script is quite straightforward to understand. The designer can redesign the room and reposition the furniture through redefining relevant variables. For instance, initially the position of the lamp (represented by the circle and hexagon) depends on the position of the table (represented by a square), when moving the table, the lamp is also moved. If the designer's view changes to have the lamp on the desk instead, the following redefinition is required:

```
/lamp/centre =  /desk/centre
```

There is hierarchical dependency in the script in Figure 3-7. A dependency network diagram (cf. Figure 2-2) can be drawn for this script to help in understanding the dependency between observables.

In some respects, grouping components by using dependency in this model is similar to the grouping feature in the MSWord drawing utility. Selected drawing objects can be grouped and treated as a single object. A dependency between the proportions of the selected objects is established so that we can resize, move and delete the group object. This dependency is limited to serve such specific purposes, whilst using a dependency in MWDS is more flexible and open.

The Room Viewer model is not just a static line drawing, but also offers scope for the designer or user to explore it interactively in an experimental way. By redefining variables and observing and interpreting the response to this interaction, he/she can understand the internal representation of the model. The fact that the internal semantic relation (cf. Figurer 2-12(a)) is easily interpreted plays a crucial role where interaction associated with the external semantic relation is concerned. The script together with its display can represent an agent perspective on the room design. The script can be viewed as representing the room as a geometric symbol and as representing atomic transformations that can be applied to the symbol. The room can be transformed through redefinition in ways that correspond closely to the idealised patterns of change associated with opening a door, or moving a table. For instance, a minor modification of the script that involves introducing a new definition of the form:

```
shape rotdoor
real angle
rotdoor = rot(door, door/hinge, angle)
```

makes it possible to model the movement of the door more realistically. The Room Viewer model has also been extended to take account of more realistic observation and interaction in other ways. For instance, 3D Room models, one featuring video-game style graphics, and the other building in laws of motion and mechanics, have been developed by Carter [Carter99] and MacDonald [Mac96] respectively (see their screenshots in Appendix B).

Variables used in the model can be viewed as representing both geometric features of the computer display and the external entities and relationships (cf. Figure 3-6). For instance, the point NW in the openshape table represents the North-West corner of the table as depicted on the computer screen in Figure 3-7 and the corner of the actual table in the external referent.

The Lines model – Pure definition-based model

The Lines model {Lines91} illustrates how a definitive script can be used to represent a complex geometrical diagram based on mathematical concepts relating to Hasse and Cayley diagrams [GM65]. The use of a definitive script in this model is different from the Room Viewer model, which is constructed with about 70 definitions to represent explicit real-world entities on the computer. The Lines model depicted in Figure 3-8 is constructed with more than 400 DoNaLD and ARCA definitive variables. Points, lines, labels and edge-coloured digraphs are defined to express sophisticated mathematical relationships being studied by Atkinson and Beynon [BYAB91] as experienced mathematicians. The motivation for constructing the four diagrams in

Figure 3-8 comes from a study of the combinatorial characteristics of simple arrangements of lines (cf. Grunbaum [Grun72]). The definitive script used in this model can be viewed as an interface mechanism[2] [BY90] to enable a mathematician to explore and visualise the complex relationships between lines in the arrangement.

ARCA is used to define complex combinatorial diagrams (viz. Cayley diagrams S4 and Poset P in Figure 3-8). It has different features from DoNaLD (cf. Section 2.1.2) since its node references are identified by indices that can be defined in terms of paths of coloured directed edges. For instance, the ARCA definition:

```
ix2 = a_S4.b_S4{ix1};
```

identifies ix2 as the index of the node in the Cayley diagram S4 that is reached from the node with the index ix1 by following first an edge of colour b_S4 (in this case, green) and then an edge of colour a_S4 (in this case, red). Thus, if ix1 is 1, then ix2 is 20 and if ix1 is 2, then ix2 is 7. In contrast, DoNaLD is used to define points and lines explicitly (cf. Arrangement A and Poset P' in Figure 3-8) without referencing abstract connections such as are associated with the edges of a combinatorial diagram.

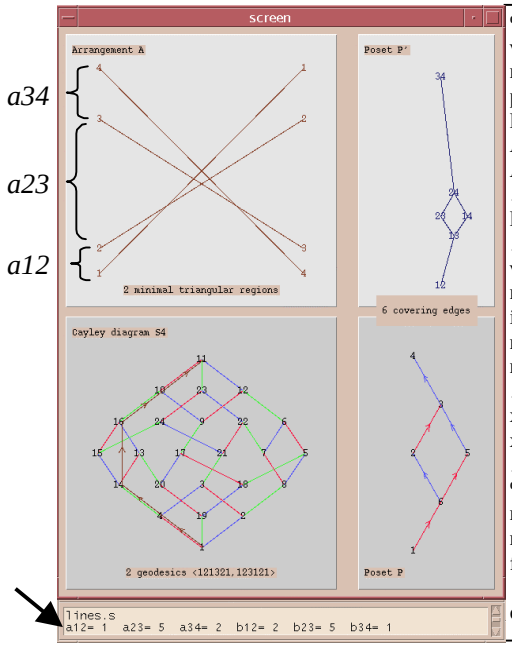
Both ARCA and DoNaLD have features that allow variables of complex type, such as diagrams and shapes, to be defined both component-by-component or using a single definition. For instance, compare the definition of the **openshape** variable door with the definition of the **shape** variable rotdoor in the Room Viewer model. Features of this nature are needed to address such issues as when it is appropriate to define the components of variables of a complex type independently. Consider, for instance, the status of the 'redefinition':

```
rotdoor/hinge = {10,50}
```

which presents a problem similar to that of updating through a view in a relational database (cf. [Denn91], p. 92). In DoNaLD, the declaration of variables as **shape** and **openshape** determines the way in which they can be defined. In ARCA, the mode of definition of variables of complex type is itself defined using an auxiliary definitive notation (cf. [Mez87]). The use of this feature is illustrated in the declaration of the variable poset of type diagram in Figure 3-8.

[2] The term 'interface mechanism' is apt because it expresses the resemblance to a mechanical linkage that reliably and instantaneously transmits state change from one place to another

The Lines model illustrates the subtlety of the visualisation process associated with the use of a definitive notation. The diagrams that appear on the screen are approximate representations of a family of lines that the script describes in an idealised manner. The implementation of DoNaLD can be interpreted with reference to an ADM artefact (cf. Figure 3-4): the points and lines in an openshape are translated by Eden definitions and associated display actions that respectively form the definition and action sets of an ADM entity. This means that the numerical values associated geometric entities are specified by formulae rather than by a procedural assignment. This gives an unusual character to the geometric variables in the DoNaLD script: they can represent idealised points and lines in the sense that the accuracy of their computer representation can be dynamically altered without compromising continuity in the perception of state.




```
%donald
viewport CONFIG
real a12,a23,a34,b12,b23,b34
point A1,A2,A3,A4,B1,B2,B3,B4
line l1,l2,l3,l4          # the 4 lines in Arrangement A
A1=Oa                  # Oa is a reference point, sc is a fixed value
A2=Oa+{0,a12 div (a12+a23+a34)*sc
…                          # A1-A4 are left-hand endpoints of l1-l4
l1=[A1,B1]                # B1-B4 are right-hand endpoints of l1-l4
…
viewport POSET         # Poset P'
real r12,r23,r34,r13,r24,r14   # Y-coordinates for 6 points in P'
int x12,x23,x34,x13,x24,x14  # X-coordinates for 6 points in P'
r12,r23,r34=a12 div b12,a23 div b23,a34 div b34
r13=(a12+a23) div (b12+b23)
…
x12=1+Z123+Z124                # Z123 defines the order of the
x13=1+(1-Z123)+Z134            # intersection of l1-l3 and l1-l2
…                              # from left to right
%arca
mode  poset = 'ab' –diag 6;
mode a_poset = col 6;
for int 0 : i = 1 to 6 do
            mode a_poset{i} = int 0
od
```

Figure 3-8: A screenshot of the Lines model and its script

Features of the model

- The definitions in the script in Figure 3-8 represent abstract mathematical relationships in terms of the complex connections amongst points and lines. In the script, there is always a counterpart for a DoNaLD variable in its internal referent on the computer display and in its external referent in the mind of the mathematician. For instance, variables A1 through A4 represent the four left-hand endpoints and B1 through B4 represent the four right-hand endpoints of l1 through l4 in Arrangement A. These points are defined in terms of six real-valued variables a12, a23, a34, b12, b23 and b34, which are key variables of the model.

The definitions of A1 through A4 and B1 through B4 reflect the interpretation of these six variables, which are such that a12, a23 and a34 (respectively b12, b23 and b34) are proportional to the distances between A1 and A2, A2 and A3, and A3 and A4 (respectively B1 and B2, B2 and B3, and B3 and B4) respectively. Changing the value of one of these key variables results in repositioning the intersection points of the four lines and indivisibly affects the display of all four diagrams. In a traditional approach to mathematical visualisation, the abstract mathematical concept is typically first expressed in terms of abstract calculation, then subsequently given a visualisation. In contrast, in MWDS in the Lines model, the analysis of mathematical concepts is closely linked to the construction of the geometric display.

- The interpretation of definitions in the Lines script is more complicated than it is for the Room Viewer model. Some definitions (such as those of l1, l2, l3 and l4) are straightforward to interpret, but some represent more abstract concepts. For instance, z123 represents the ordering of intersection points between the lines l1, l2 and l3. The interpretation of this definition is: z123 = 1 if and only if, reading from left to right, l1 intersects l2 before l1 intersects l3.

- A group of definitions can represent a specific concept required in the model. For instance, the variables A1 through A4 and the variables z123, z124, etc. represent two such groups. We can generate generic definitions and interpretations for such groups, as discussed in [Rung97]. Alternatively, the Lines script can be interpreted based on its dependency network, so that the hierarchical relationships between variables are displayed. For instance, X23 is defined with reference to other variables as follows:

```
                    X23 = 2-z123+Z234
z123 = if r12<r23 then 1 else 0        Z234 = if r24<r23 then 1 else 0
r12 = a12 div b12                      r24 = (a23+a34) div (b23+b34)
r23 = a23 div b23
```

  All of the key variables will appear at the leaves of such a dependency network diagram. Different ways of organising the script can reveal diverse interpretations.

- Experiments involving the extension and elaboration of scripts play a major role in the development of definitive models. A script is incrementally redefined and modified whilst observing how state changes in a stimulus-response manner. Extensions and modifications corresponding to different views can be made at any time. For instance, a new view can be

introduced into the Lines model that extends the user interface to assist comprehension of the external semantic relation by introducing dynamic annotation (cf. Figure 7-8).

- The Lines model was originally developed for arrangements of four lines. It is difficult to display the appropriate Cayley diagram when the number of lines is greater than 4, but the mathematical concepts and relationships behind the model can apply to arrangements of arbitrary size. As explored in [Rung97, ModelWeb], generalisations of the Arrangement A and Poset P' diagrams have been constructed and used to study arrangements with up to 9 lines. The details of the technique used for script generation will be discussed in Chapter 7.

The Lines model has illustrated how to use definitions (with no procedures or functions) to represent line drawings in which their patterns of change correspond to the abstract mathematical concept. The interpretation of the internal semantic relation (cf. Figure 2-12(a)) involves understanding the mathematical concept. A group of definitions is regarded as a passive agent, which acts through the propagation of state changes. Observable is used to represent a referent (cf. External entity-relationship model in Figure 3-6) in relation to the mathematician's conception and observation. This is different from a traditional approach that typically makes use of a control loop to manipulate with variables in calculating the output.

Summary of Section 3.2.1

A comparison of the Room Viewer and the Lines models highlights the need for radically different kinds of internal and external semantic relations. The interpretations of the DoNaLD line drawing as a room in Figure 3-7 and of the ARCA Cayley diagram in Figure 3-8 engage the modeller's mind in totally different ways. In the external referent for the Room Viewer model, the key observables are physical real-world objects and their attributes. In the external referent for the Lines model, the key observables are abstract mathematical relationships amongst idealised geometric entities. These two kinds of external semantic relation demand different qualities of the internal semantic relation. In effect, the computer has to play the role of different instruments in supporting different mental activities, and this motivates the design and use of different definitive notations.

3.2.2 Comprehension of state in multi-agent modelling

In this section we contrast two ways of using an ADM artefact to model a multi-agent scenario with respect to two definitive models concerned with classroom interaction and railway accident

scenarios. The modeller projects him/herself into different agents' roles involving in the situated model (cf. Figure 3-5).

The Classroom Interaction model – simulation

The Classroom Interaction model {Class95} has been developed to simulate the interaction between pupils and the teacher in a classroom, so that student teachers can use it as a simulation to learn about the behaviour of pupils. The factors involved in the interaction between the teacher and the pupils and amongst the pupils themselves were studied and analysed by Emma Davis [Davis95] in consultation with Steve Russ and Sean Neill, an educational psychologist at the University of Warwick. The pupils' behaviour is to be affected both by the interaction given by the user (who acts as a teacher) and by fellow class members [Davis95]. The factors that influence the pupils' behaviour include the personal values and personality of the pupils and the social behaviour of each pupil. The model offers the feature that the student teacher can experimentally assign different values to the parameters that are deemed to affect the behaviour of each pupil and observe his/her behaviour, as depicted in Figure 3-9.

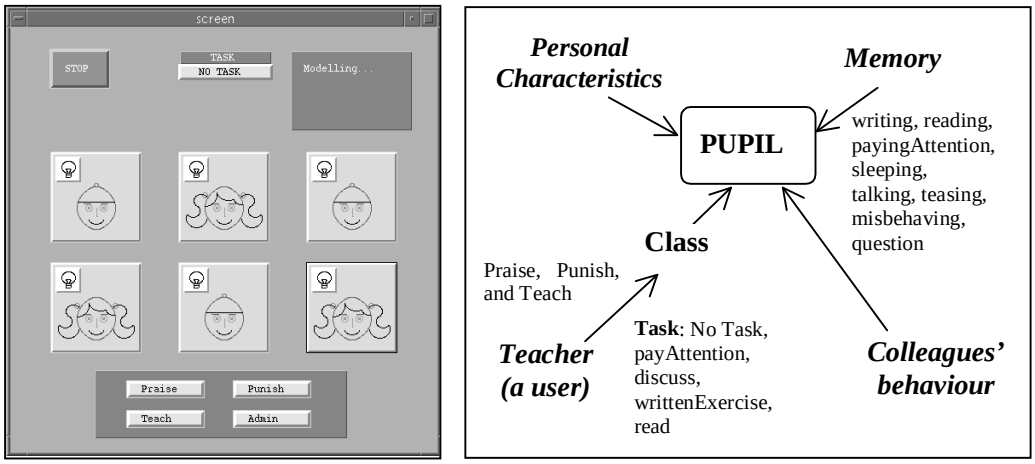


Figure 3-9: A screenshot of Classroom and pupil's behaviour decision diagram

Features of the model

- Definitions in the script serve to represent the state of mind of the teacher and of each pupil and the relations, such as the influence of the teacher over the pupils, the influence of the pupils over the teacher and the influences of surrounding pupils on each pupil. Through redefinitions and triggered actions, the pupils' actions are simulated concurrently. The mediation of interaction by definition is different from an object-oriented approach that uses message passing as a communication medium between objects.

- The Classroom Interaction model is explicitly implemented as an ADM artefact. The two main agents in the model are the teacher and the pupils. The role of the teacher is played by the user. Two main ADM entities are introduced: Pupil, which represents a pupil agent, and Class, which represents the classroom environment. In the model, there is one teacher but several pupils (cf. Figure 3-9). The diagram in Figure 3-9 summarises the factors that determine the pupil's behaviour. These include the kind of instruction currently in progress (e.g. No Task, pay Attention etc.), the current activities in which pupils are engaged (e.g. writing, reading etc.), the responsive action from the teacher (e.g. praise, punish) and the set of pupil's personal characteristics (e.g. name, ability, sociability, risk taking etc.). The teacher can manually assign a value in the range 1 to 10 to each characteristic of the pupil.

- The LSD account in Listing 3-1 describes the observables required for each agent and their classification. The modeller projects herself on different agent roles to specify such observables (cf. Figure 2-19). The LSD account for the Agent Pupil in Listing 3-1 can be interpreted as stating that each Pupil has *state* observables that capture his/her properties, such as name, ability and activity and *oracle* observables that represent factors under the control of other agents, such as instruction and teacherActivity. Based on this LSD account, the adm specification in Listing 3-2 is developed. The entity Class in Listing 3-2 determines the general environment of the classroom, including the interfaces through which the teacher interacts. The entity Pupil describes the definitions and actions used to model a pupil's behaviour. This adm definition is automatically translated into Eden script by a translator developed by Y. P. Yung and P-H. Sun so that the model can be run using the tkeden tool. As discussed in Section 2.2, an LSD account is very broad and open-ended to interpret. The adm definition in Listing 3-2 is just one example of how the account can be interpreted in the adm.

- A feature of the Classroom Interaction model is the use of mechanisms based on cost-benefit analysis in constructing the behavioural model of the pupils. Part of the motivation for the case study was to investigate whether classical models of animal behaviour [Daw95] could be applied in the classroom context.

```
Agent Pupil{
STATE
        name
        ability
        activity
        pupilMemory
        subjectInterest
        sociability
        riskTaking
        attentionSeeking
        memLength
ORACLE
        instruction
        teacherActivity
        teacherObserving
        colleaquesActivity
HANDLE
        activity
        memory
}
```

```
Agent Teacher{
STATE
        activity
        observing
        awareness
        strictness
        memLegth
        teacherInstruction
        teacherMemory
ORACLE
        pupilActivity
DERIVATE
        pupilCharacteristics
        numberWorking
HANDLE
        location
        activity
        memory
}
```

Listing 3-1: LSD account for Classroom Interaction

```
%adm
entity Class(){
definition
      numActivityAvail=8, classID=0, classSize=6, ...
action
      iClock==taskCompletionTime print("First class guard")
      -> task=noTask; payAttentionColour="darkSeaGreen1"; ... iClock==...
}
entity Pupil(_id){
definition
      name{_id}=_id, activity{_id}=payingAttention, conscientiousness{_id}=medium,
      subjectInterest{_id}=medium, ...
action
    (((teacherActivity==praise)||(teacherActivity==punish))&&(teacherObserving==[classID])&&(classActivity!=0))
    -> pastResults{_id}[|classActivity|]=insertAtFront(pastResults{_id}[|Activity|],[|teacherActivity|],,e,Length{_id});
    ...,
   (((teacherActivity==praise)||(teacherActivity==punish))&&(teacherObserving==[classID])&&(classActivity==0))
    -> ...
}
startClock=0; state = "atStart";
...
Class(); Pupil(1); Pupil(2); ...
```

Listing 3-2: The adm definition for Classroom Interaction

The model illustrates how – in principle – the roles of human agents can be embodied in an ADM artefact. The way in which interaction is represented by a set of definitions together with a set of actions in the Class and Pupil entities is illustrated in Listing 3-2. The transitions that result from agent action are triggered by redefinitions. The actions performed by one agent may affect others' behaviours through propagation. The LSD analysis is used in this model to aid the modeller to interpret and document observables from an internal agent viewpoint and the adm definition is then derived from the LSD account.

The Railway Accident model – Distributed definitive model

The Railway Accident model {Rail99} illustrates how it is possible to distribute different observational viewpoints to several workstations. Definitive scripts can be sent across the network and each machine maintains its state by running its own dependency maintenance system. The model enables us to simulate a railway accident that occurred in the Clayton Tunnel in 1861. Our simulation is based on an account by Rolt in [Rolt82] and is summarised in Box 1 in Appendix B.

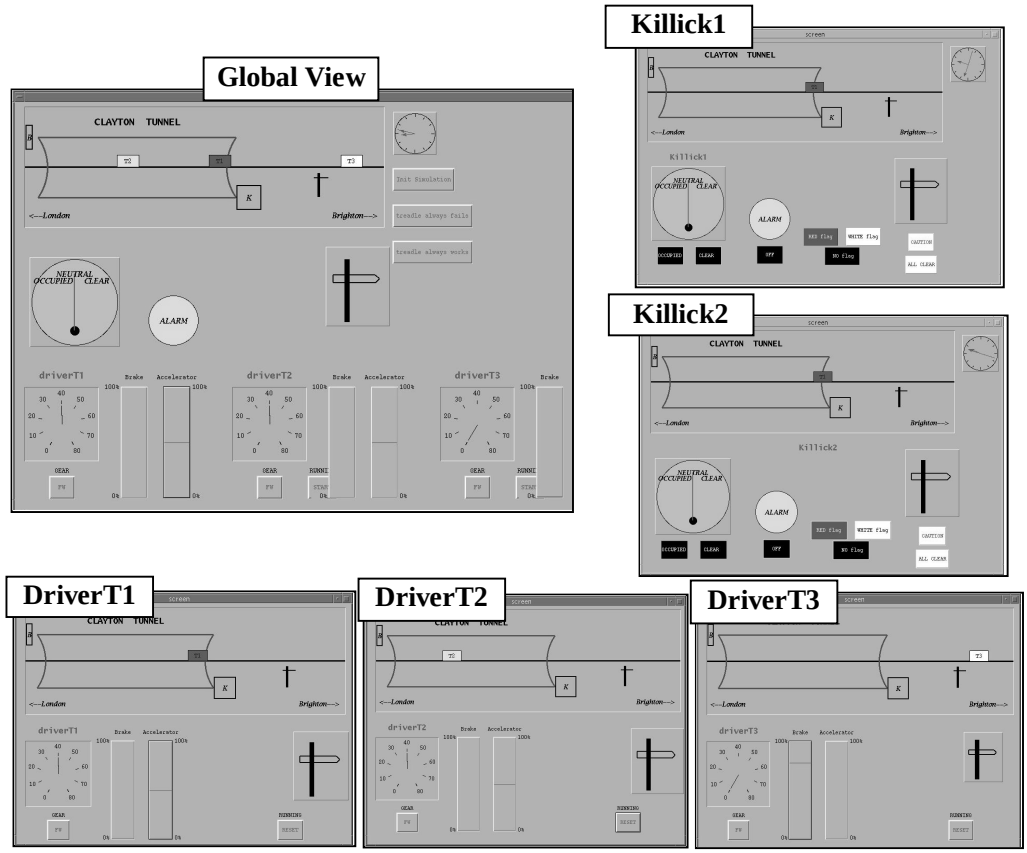


Figure 3-10: Screenshots of the Railway Accident model running on different machines

The perspectives of the key personnel involved in the accident scenario, together with a global view, are displayed in Figure 3-10. Each person is modelled as an LSD agent following an LSD account for the model that was developed by Sun [Sun99] to capture and classify each agent's observables and take account of issues of perception. A set of definitions and actions is developed to represent the role and viewpoint of each of the agents. As depicted in Figure 3-11, what each agent observes is influenced by many factors such as the position of an agent and the position of trains etc. relative to the observer. The agents can interact with each other and their environment by redefining the variables which they are privileged to change. Redefinitions are transmitted across the network and, as a result of the automatic updating caused by redefinitions,

the state relating to each agent changes and the visualisation reflecting the agent's view is redisplayed.

There are typically discrepancies between the viewpoints of the different human agents due to delays in the transmission of definitions. The global view, which is maintained at the server of the client-server network, is updated to reflect the state-changing actions performed at every client workstation so that, in some sense, it can be construed as giving a comprehensive picture of the overall state. This is one view of what comprehension of state might mean, but it is inadequate when all the different – possibly inconsistent – viewpoints are taken into account. An alternative view is that of an external observer of the entire distributed model. In some sense, this is the more authentic notion of comprehension of state for the railway model.

Features of the model

- In executing the model, the roles of the various agents can be played by different human participants at the appropriate workstations. For instance, with reference to Figure 3-11, the roles of the drivers of the trains T1, T2 and T3 depicted in the global view are played by the participants DriverT1, DriverT2 and DriverT3 respectively, whilst the role of the signalman Killick is shared between two participants Killick1 and Killick2. Each participant interprets the situation and takes action according to the state as he/she observes. For instance, Killick1 will set the telegraph to 'occupied' when he/she sees a train entering the tunnel and will determine the precise time at which the signal is sent.
- Sun's LSD account on which the model is based reflects realistic common sense assumptions about the capabilities of the agents in the accident scenario. For instance, Killick cannot see the speedometers of the trains or the state of the telegraph in the signal box at the opposite end of the tunnel, and can only change the status of specific artefacts associated with his signal box. There is a subtle relationship between these capabilities and the capabilities of the participants who play the roles of these agents. For instance, the visualisation of the model is such that a train disappears from the screen instantly when it enters the tunnel, whereas in reality its presence might still be evident from smoke and noise. Factors such as the speed of reaction of the participants might also be more or less realistic. The physical layout of the workstations in the distributed simulation is also significant. For instance, in practice, the workstations have typically been configured in such a way that the participants have more knowledge of the global situation (e.g. about the locations of trains and the intentions of

other agents/participants) than is realistic. An alternative way to exercise the model would involve placing the participants in different rooms to eliminate some unrealistic elements in the communication.

Summary of Section 3.2.2

The Railway Accident model demonstrates different aspects from the Classroom Interaction model since it exploits 'participatory theatre' (as discussed in connection with Agentsheets in Chapter 1) by distributing scripts and scenarios amongst players so that they can decide when and how to act. The role of human decision-making and interaction is significant. In contrast, the Classroom Interaction model is more concerned with simulating autonomous agents interacting with each other. Only a single user is involved in interacting with the model (i.e. changing the value of parameters and observing the effect). Taking together, the two models illustrate a wide variety of perspectives on comprehension of state.

3.2.3 The internal semantic relation

A major theme of MWDS is using a script to represent state as experienced or perceived by the modeller. The modeller can take the role of an external agent (such as a designer or user), or project him/herself into the role of an internal agent, perhaps to implement it automatically. Observables always reflect their internal counterparts in the model, external counterparts in the situation or both. The discussion of the two definitive models in this section will demonstrate the use of definitions to represent user data (the Car History model) and actions to represent user interaction (the Generic User Interface model) from the modeller's perspective.

The Car History model

The Car History model {Car94} is one of the most elaborate definitive models built to serve an application. It combines data storage similar to that conventionally recorded in a relational database such as ORACLE with temporal information. The model is concerned with recording two kinds of data about cars:

- abstract data, such as possible car types, car brands and details of which parts are compatible with a given model of car;
- data about specific cars owned by the user, such as registration plate numbers and details of when particular car parts are changed, as this develops over time.

The discussion below focuses on the data model used to record the abstract data about cars. This relates to the six pre-defined cars (viz. Ford_Escort, Ford_Orion, Ford_Fiesta, Ford_Granada, FIAT_Regata and FIAT_Uno) shown in Listing 3-3. Each car can have a different style: saloon, estate, convertible, sports and van. The complete model provides features for the user to record the history of each car they have owned, from its first purchase to its current status. The user can interact with the model in various pre-defined ways, for example, to view a car, to create a new car with different options (e.g. with a new registration number or with an old number plate) and to record all changes in a log book. Definitive scripts are used to capture the data and their relationships, and to control the visualisation of the model.

Features of the model

- The script used to represent data in this model serves a similar function to a database. A simple list data structure is used to record all the data in a way that generalises the table in a relational database system (cf. Figure 3-11). The variable cars in Listing 3-3 resembles a table in the database that contains six car types (cf. Figure 3-11), each of which is represented by using the name of the car-type (e.g. "Ford_Escort") in conjunction with the Eden list variable identified by this name (e.g. Ford_Escort). The same pattern of script is also used in recording data about models, exhausts and engines. Representing the data with definitive scripts in this way allows flexibility to add relationships between data. For instance, we might use the definition:

  fe_exhaust is fo_exhaust;

  to indicate that the Ford_Escort and the Ford_Orion have the same exhaust.

```
cars is ["Ford_Escort", Ford_Escort, "Ford_Orion", Ford_Orion, "Ford_Fiesta", Ford_Fiesta,
        "Ford_Granada", Ford_Granada , "FIAT_Regata", FIAT_Regata, "FIAT_Uno", FIAT_Uno];
Ford_Escort is ["model", fe_model, "exhaust", fe_exhaust, "engine", fe_engine, "rest", fe_rest];
Ford_Orion is ["model", fo_model, "exhaust", fo_exhaust, "engine", fo_engine, "rest", fo_rest];
Ford_Fiesta is ["model", ff_model, "exhaust", ff_exhaust, "engine", ff_engine, "rest", ff_rest];
fe_model is ["Mk1", Mk1, "Mk2", Mk2];
Mk1 is [ ["saloon", [1,1,1,1,1,1,1,1,1,1,1] ], [@], [@], ["sports", [1,1,1,1,1,1,1,1,1,1,1] ], [@]];
```

Listing 3-3: Script to record details of cars

- As shown in Listing 3-3, each variable such as Ford_Escort, fe_model and fe_exhaust represents data with reference to other variables in a hierarchical manner. For instance, with reference to Figure 3-11, if we want to look for the components of Fe_BOSAL, which represents one option for a Ford_Escort exhaust, we need to consult the value of feB_clps and feB_mnt as well. The name of the variable refers to the sort of data that it represents. Data represented in this form is open to extension in the same way that we can modify the

file structure in a Unix system. Variables can be assigned to depend on other variables so that the structure of the data dependency changes.

Another distinctive feature of the script in this model is that the data stored in the variables also defines the visualisation of the model of a car. For instance, the value of the variable Mk1 in Listing 3-3 is the list [1,1,1,…,1] which defines the line drawing for this car type and style, as depicted on the left in Figure 3-12.

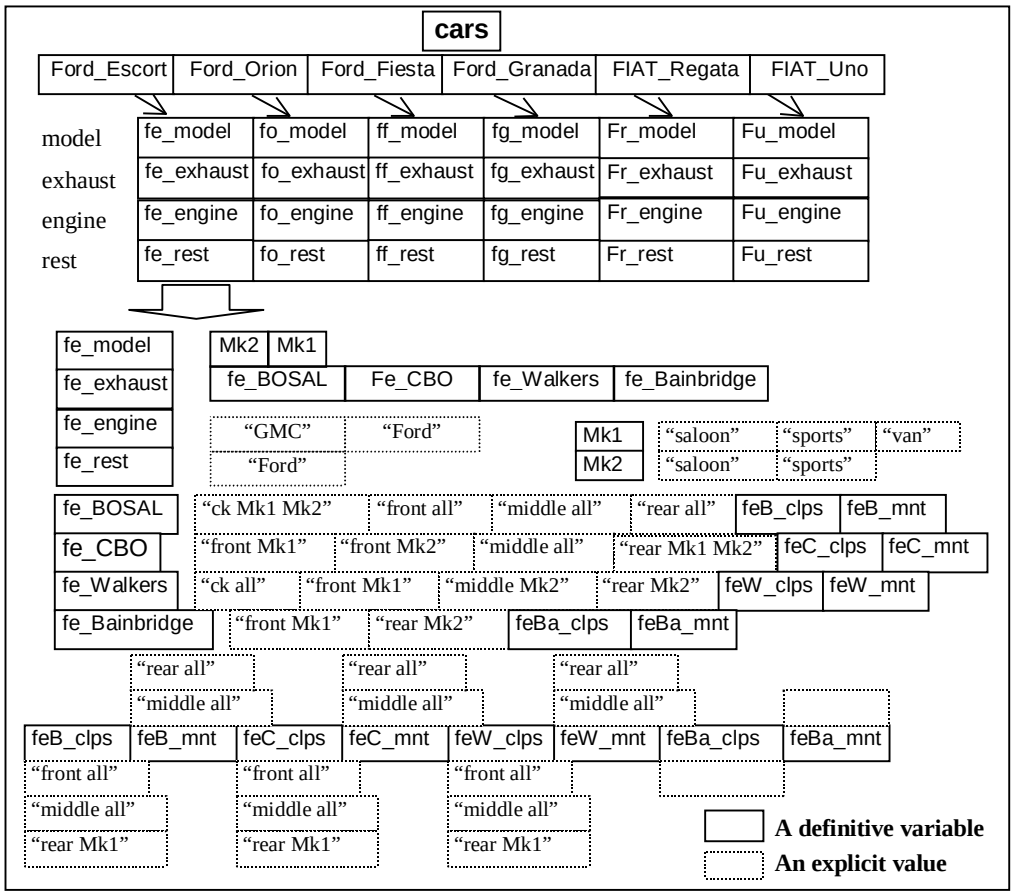


Figure 3-11: The data representation diagram for a specific car (i.e. Ford_Escort)

- The structure of line drawings for displaying the car with different styles such as saloon, sports and convertible is very distinctive. The DoNaLD openshape car is defined, as a template, so that the display of various car styles is determined by changing the value of definitions in the openshape. By redefining some variables (representing properties of the car such as wheelnut, cartype and door), the display of the conventional car is changed to the convertible car, as depicted in Figure 3-12. This technique is similar to the use of templates in PIC [Kern82], and to the use of parametric geometric modelling tools [SM95]. The script can be exercised in isolation from the model. It can also be reused in other models. The parameters used in the script for displaying a car are as shown in Figure 3-12.

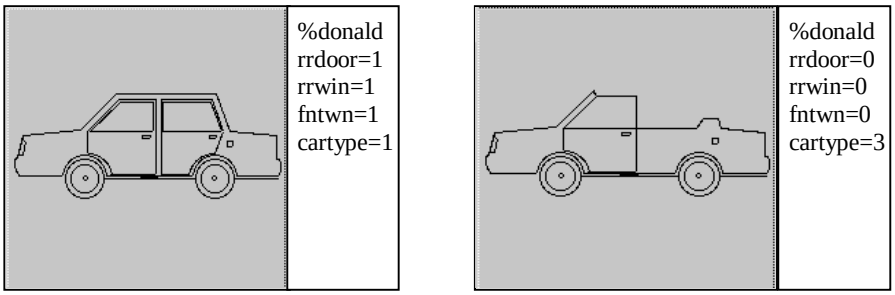
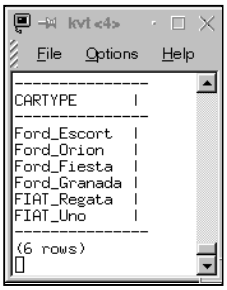


Figure 3-12: Transforming a car from a saloon to a convertible

- The manipulation and ordering of display windows is based upon the variable layout, a triggering variable that records the list of windows in the current display. When the value of layout changes, the display of the windows will be updated automatically with reference to this list.

The Car History model illustrates very rich data organisation. The record of the car, the display of the car and the data store for the car are defined using complex inter-related variables. Using definitive scripts to represent the data and its data structure illustrates a flexible and open-ended means for defining the data to reflect the modeller's viewpoint. The structure of the data can be adapted easily to suit the user's needs. For instance, the lists that define fe_model and fe_exhaust can be extended to take account of the arrival of a new model or of an additional supplier for an exhaust.




```
%eddi
CARS(CARTYPE char);
%eden
func projcar{
        para carlist;
        auto result = [];
        while(carlist!=[]{ result // [ [carlist[1]] ]; shift(carlist); shift(carlist); };
        return result;
};
cars is ["Ford_Escort", Ford_Escort, ... ];
CARS0 = CARS; CARS is CARS0 // projcar(cars);
```

Figure 3-13: The use of Eddi in the Car History model

The Car History model was developed prior to the implementation of the definitive notation Eddi for relational tables. No modification of the data model other than the addition of the script listed in Figure 3-13 is needed in order to make use of Eddi to display the data store for a car in a tabular form. This enables us to use relational queries to help in understanding and interpreting the data.

The Generic User Interface model

The Generic User Interface model {GUI96} is an application that features the agent-oriented construction of user interfaces. The modeller aimed to exploit the characteristics of definitive scripts as a framework for agent action in constructing user interface applications. The principles behind this use of MWDS are illustrated in a simple form in Agentsheets [AgentWeb], where agent actions take the form of redefinitions in a spreadsheet-like space of cells (cf. Figure 1-4). In such a model, simple actions can generate rich behaviours because of the many different interpretations that a redefinition can have in the context of different definitive scripts.

Two generic user interface applications are developed, both of which have loose specific goals (as is part of the nature of an ADM artefact). The themes behind these two applications are 'timeline record' and 'form design'. They are very open-ended in character and can be used (or modified for use) in diverse applications. These two models illustrate how the principles of MWDS can be elaborated to extend the expressive power of the Scout definitive notation to include features similar to Visual Basic [DDN99] and Java Swing [ELW98].

Features of the Timeline Record model

- The Timeline Record application was originally developed with a view to applying MWDS to the maintenance of historical records relating to the artefacts (pictures, photographs, documents and objects) in a museum or library. A screenshot of the application is depicted in Figure 3-14. The three rectangular regions in the display represent timeline records on which the user can specify time intervals, each represented by a bar on the record, by using the mouse and menu buttons in combination. A file of information can be associated with each time interval. The application allows the user to define timelines, specify and modify the time intervals, and edit and retrieve the information in the associated files.

- In the model, several automatic internal agents involved in managing the user interaction with the application are identified (cf. Figure 3-14). These agents serve characteristic roles; they include *Select* (the recorder of the user's options), *Sensor* (the sensor on a timeline record), and *Info* (the reporter of the information). Each of these agents is implemented in a different way. *Select* is associated with the bank of mode buttons. *Sensor* is associated with the sensitive Scout windows representing timeline records. *Info* is associated with the Eden showinfoscr action listed below. *Select* sets the observable mode according to button selection. *Sensor* returns information about the timeline record, the position and the index of

the bar with which a mouse click is associated. The way in which these agents operate reflects the modeller's perspective on the user's requirements, and is characteristic of the generic user-interface application.

By way of illustration, in the current state of the application depicted in Figure 3-14, the INFORMATION button is selected so that the value of the variable mode is set to info. When the user clicks on a timeline record, agent *Info* is invoked by the triggering variable record. In fact, all agents (e.g. *NewRec*, *New* etc.) are invoked but only the agent *Info* takes action because of the value of mode. The agent *Info* will consult the agent *Sensor* associated with the selected record and bar to determine which file is being selected by the user. The following script defines a triggered action for the agent *Info*:

```
proc showinfoscr: record{
        if (mode==info){
                if (record==prs1) ftomr1(inr1()); else
                if (record==prs2) ftorm2(inr2()); else
                if (record==prs3) ftomr3(inr3());
        }
}
/* prs1 indicates that the mouse is pressed on timeline record 1 etc. */
```

In the script above, the procedure ftomr1() (respectively, ftomr2() and ftomr3()) retrieves the file associated with the bar indexed by inr1() (respectively, inr2() and inr3()) to determine the right information to display. A similar technique is used in constructing the other agents (e.g. *New*, *NewRec* etc.).

- The model illustrates a distinctive technique for using a definitive script to update the DoNaLD script that represents a bar. Such updating is used when introducing a new bar and moving or resizing an existing bar. The technique is exemplified in the following script:

```
proc r1drawbox{                         /* procedure for manipulating a bar on timeline record 1*/
   auto x, l
     x = $2;                        /* the position on the record of the bar selected by the user */
     l =  $3;          / * the offset of the mouse click in the bar: default value length of the bar */
     barnswno = $1;
     barnlenno = $1;
     `barnsw`[2] = x; `barnlen` = l;          /* the back quotes ( " ` " ) are used to convert */
}                                                                       /* a string to a variable name*/
barnsw is "_bar"//str(barnswno)//"_SW";
barnlen is "_bar"//str(barnlenno)//"_Length";
```

The definitions barnsw and barnlen are used as templates to hold a set of Eden counterparts (e.g. _bar1_SW, _bar2_SW, _bar1_Length) for DoNaLD variables with similar name. When the values of barnswno and barnlenno change, the values of the strings barnsw and barnlen are also changed. The back quotes (cf. Section 2.1.2) are used

to turn string variables to Eden variables. For instance, if barnswno has the value 1, the value of the variable _bar1_SW will be assigned to the value of x, and the DoNaLD line bar1 will be displayed with this corresponding value.

The action of the agent *Move*, as it applies to bar 1 on the timeline record 1, illustrates a principle similar to that used in Agentsheets. The counterpart of the dependencies in an agentsheet are expressed in the definitions of the DoNaLD script for the openshape bar1. The agent action takes the form of a redefinition, in this case, of the South-West corner of bar 1, upon which the location of the bar depends. The procedure r1drawbox() generates the appropriate redefinition according to which bar and what option has been selected by the user. The agent move can be given a simple LSD specification: it has as its oracles the observables the current mode, record index, bar index and mouse press and release positions and has the South-West corner of the appropriate bar as a handle.

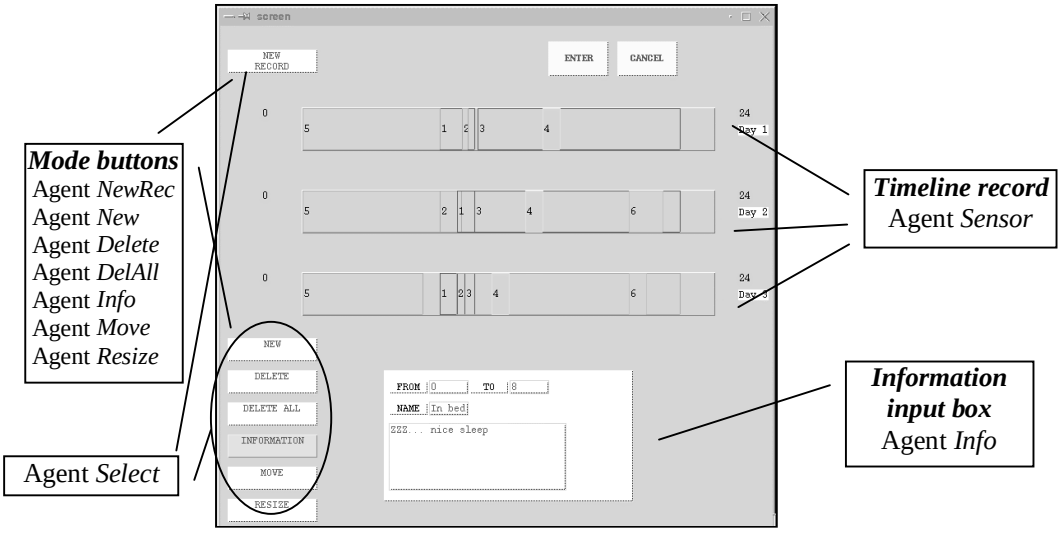


Figure 3-14: A screenshot of the Timeline Record model

- The model can be used in a variety of ways. It can serve as a diary, as depicted in Figure 3-10, or it can be used as an interactive reference to record the history of Kings and Queens of England, as discussed in [ModelWeb]. The model can be easily modified to work in conjunction with other models. For instance, a timeline could be used to record the display mode of the Digital Watch model (cf. Figure 3-17) over a period of time.

Features of the Form Design model

- The Form Design model was developed after the Time Record model and its implementation was based on reusing the techniques for agent construction discussed previously. This

application is simpler than time recording. The internal agents in this case represent text boxes, a picture box and mode buttons. Each text box can be moved around the design space and text written on each box can be assigned to be fixed text (i.e. not editable) or data text (cf. Figure 3-15(a)). We can also specify whether text is to be interpreted as numerical or string data.

- In this application, on selecting option **i** mode, and then selecting a text box, the variables in the defining script for the text box, together with their current values, are displayed in the panel below the design space (cf. Figure 3-15(b)). This gives access to meta-information about the text box. This feature enables the user to point and click at the screen to gain information about the implementation of the text boxes. This extension is similar to special-purpose script extension such as is discussed in Section 7.2.1.

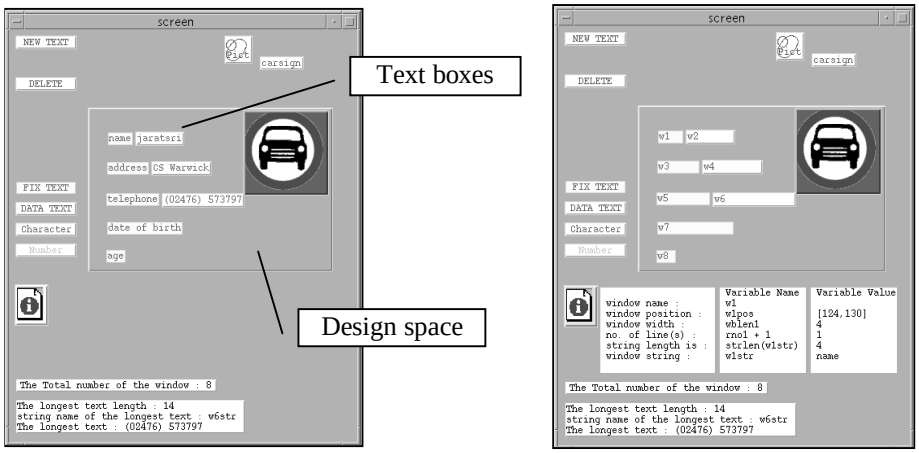


Figure 3-15: (a) Screenshots of the form design application, (b) using i option

- As in the Timeline Recording application, this application can be adapted for use in a variety of ways. For instance, by including a small additional script, the application can be used to translate a basic vocabulary or to check the spelling of words [ModelWeb]. It can also be used to monitor how many times the user clicks on each option button. By way of illustration, introducing the following script will count the number of times that the user select the 'FIX TEXT' option:

```
fprs=0;
proc cntfprs: fixbut_mouse_1{
        if(fixbut_mouse_1[2]==4){
                fprs = fprs+1;
                w2str = str(fprs);
        }
}
```

Summary of Section 3.2.3

The Generic User Interface model demonstrates different aspects from the Car History model in representing the internal components. In the Car History model, variables and dependencies are used to represent the data and their relationships. The major concern is on structuring the internal data-dependency model, in this case the car database. In contrast, the Generic User Interface model focuses on the representation of the automatic internal agents required in user-interface applications. With reference to the structure of an ADM artefact (cf. Figures 3-5 and 3-6), each internal agent is represented by an ADM entity: a set of definitions together with a set of actions.

3.2.4 Open and closed interaction

This section illustrates two contrasting motivations for building an artefact. The Monotone Boolean Function model has been developed to support research into planar monotone circuits, and the interaction throughout the model building has had an open-ended and exploratory character. The model is unfinished and further extension and adaptation is envisaged in conjunction with projected future research. The Digital Watch model was originally developed from the statechart representing its interface introduced by Harel in [Har92]. It illustrates the use of MWDS to model a device as an ADM artefact (cf. Chapter 5); the model is designed with reference to pre-specified and well-analysed patterns of state change. The model is also 'complete' in the sense that all the functionality associated with normal use of a digital watch is captured. The presentation of the ADM artefact is also such as to promote closed interaction, even though the scope for open interaction still exists.

The Monotone Boolean Function (MBF) model

The MBF model {MBF99} has been developed to visualise an algorithm that determines whether or not the input MBF[3] is planar computable and, if it is, construct an appropriate planar circuit as depicted in Figure 3-16. The criterion for testing planar computability and constructing the circuits was introduced in [BB87], from which the algorithm used in the MBF model is drawn.

The model can be divided into two complementary parts. We first determine whether or not a given MBF is planar computable. Where appropriate, we then compute and draw the corresponding circuit for it. To determine whether a given MBF f is planar monotone

[3] A monotone boolean function is a logic function containing only 'and' and 'or' logic gates.

computable, the function *f* has to be transformed into its conjunctive[4] and disjunctive[5] normal forms. These two normal forms are computed using two functions, the Prime Implicant Function (PIF) and the Prime Clause Function (PCF). Visual representations of these normal forms, as they apply to the MBF:

$$f = (a \wedge (b \vee (c \wedge d)) \vee (b \wedge d \wedge e)$$

appears in the top left hand corner of Figure 3-16.

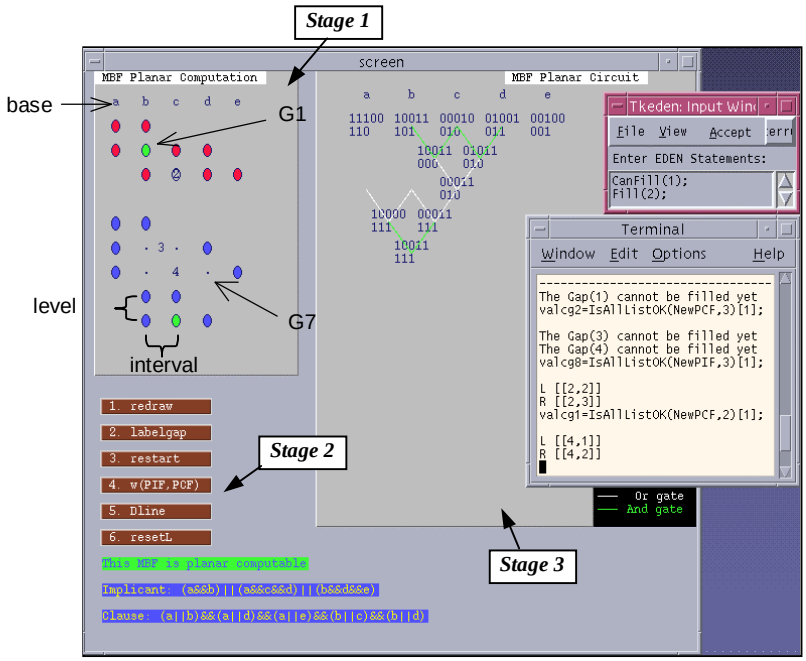


Figure 3-16: A screenshot of the MBF model

The essential principle of the algorithm to determine whether *f* is planar monotone computable involves introducing fragments of planar monotone circuit such as that depicted in Figure 3-16. Provided that the gaps in the visual representation of PIF and PCF – as indicated by the numerical labels in Figure 3-16 – are appropriately configured, the introduction of such a fragment generates new boolean functions to replace the original inputs that are at least as computationally useful in that they can stand in for the input in any computation that returns *f*. The construction of such functions to improve upon the original inputs is indivisibly associated with 'filling the gaps' in the visual representation. An MBF is planar monotone computable if and only if all gaps in the visual representations of PIF and PCF can be filled in this way.

[4] A function is written in form of $\wedge(x(\vee x)*)$

Features of the model

The criterion for planar monotone computability of a given MBF *f* can be expressed using a definitive script of the form:

```
Allset is initAllset(f); evalAllset is findTF(Allset);
ClauseF is collectTF(evalAllset, "F"); PCF01 is subset(ClauseF, 0);
ImplicantT is subset(evalAllset, "T"); PIF01 is subset(ImplicantT, 1);
PIF is BitToIndex(PIF01,1);  PCF is BitToIndex(PCF01,1); planar is IsPlanar(PIF, PCF);

/* f is a given MBF and Allset initiates a list for all possible permutations for f. ClauseF and ImplicatT
        represent the sets of intermediate values for computing PIF and PCF */
```

Listing 3-4: The script for computing whether a given MBF is planar computable

The visual representations of PIF and PCF are derived from this script by adding appropriate Scout and DoNaLD definitions.

The circuits for the planar computable MBF can be constructed dynamically according to the choices of gap to be filled as specified interactively by the user. Figure 3-16 illustrates how the circuit is constructed on the basis of these choices. In the figure, the user has already filled gap 5 and 1 in that order, and is just about to fill gap 2. The set of definitions for the line drawing of the circuit is generated dynamically corresponding to the user's input.

The execute("…") command (cf. Section 2.1.2) is used to generate and interpret scripts dynamically. For instance, the following action illustrates how a script to declare and define the points that lie in the gaps (cf. Figure 3-16, where there are eight such points G1, G2, …, G8) is generated and passed to be executed:

```
proc FillGap: fillgap{
    if (fillgap==1) {
        execute("%donald\n"//DeclarGap());
            /* DeclarGap() generates a declaration for the points G1, G2, … */
        DonGap=genDonGap(GapPIF,GapPCF); execute(("%donald\n"//DonGap); …
            /*genDonGap() generates definitions for G1, G2 */
                    }
}
```

The above triggered action generates definitions of the following form:

```
point G1, G2, G3, G4, G5, G6, G7, G8
G1 = {(pointG1! –1) * interval, level*2} + base
G2 = {(pointG2! –1) * interval, level*3} + base
```

[5] A function is written in form of $\vee(x(\wedge x)*)$

where pointG1! refers to an Eden variable that represents the index of the input variable that is vertically aligned with the point G1, and the observables interval, level and base are as indicated in Figure 3-16.

The MBF model is a study of what is possible in principle and illustrates aspirations for the modelling that are currently beyond reach. It is a partially successful attempt to integrate model-building with the kind of open-ended and exploratory activities that occur in mathematical research. The author – as the model builder – did gain understanding of the underlying algorithm through developing the model. Before MWDS of this nature can be usefully integrated with mathematical research, building the model would have to be significantly easier than conducting the research without the computer support.

During the development of the model, some problematic issues with the current definitive tool emerged. For instance, it is hard to eliminate definitions from the model once they have been introduced. Some definitions representing visual elements still exist in the model, but they are made invisible by displaying them outside the current view.

The modeller adopted various roles as an external agent in constructing the model. She first built an initial model to understand the algorithm to determine planar monotone computability. This involved developing the script in Listing 3-4, in which there is no visualisation, in an exploratory manner. She later took up the role of the designer of an application to help the user to understand how the circuit constructed based on the choices of the gaps to be filled. This involved adding the visualisation at the top left of Figure 3-16 (Stage 1) and the user interface below (Stage 2). Her perspective then shifted to developing automatic agents to dynamically generate the display for the circuit and to support the richer interaction needed (e.g.) to determine which gaps can be filled and to specify which gaps to fill (Stage 3). Note that this interaction is supported through the Eden input window (cf. Figure 3-16) and not by a graphical user-interface. The model building illustrates a typical progression from manual interaction through the input window interface to graphical user interaction with automatic internal actions.

The Digital Watch model

The Digital Watch model {Digital92} depicted in Figure 3-17 illustrates how definitive scripts can be used to model a device. As Figure 3-17 shows, the model has two components. The right hand side of the figure displays the actual physical behaviour of the digital watch together with

an analogue clock. On the left hand side is the display of the statechart that describes the principal functionality of the watch. The user can interact with the digital watch (on RHS) and observe the way in which this interaction is interpreted in the statechart (on LHS). For instance, Figure 3-17 displays a situation in which the actual time is just after 1:45 PM (as indicated on the analogue clock) and the user is in the process of setting the alarm. The statechart is accordingly in the *t-min* state within the *up-alarm* state within the *displays* state.

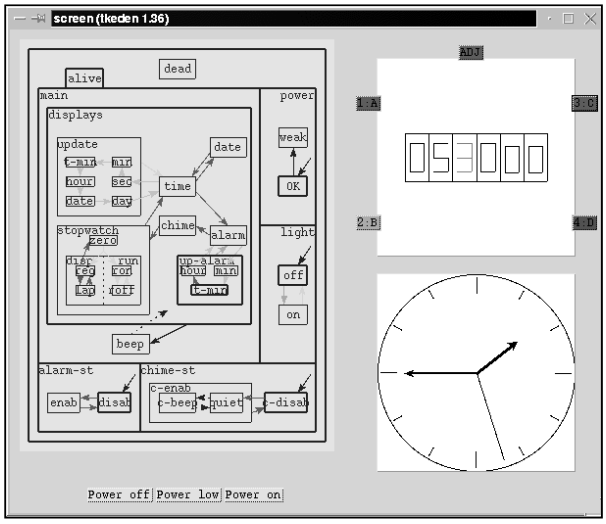


Figure 3-17: A screenshot of the Digital Watch model

Features of the model

- The model can be useful in the design of the mechanism of the digital watch in that the designer can have a user's view during its development. When designing the watch, both the internal mechanism and external output for the user need to be understood. Definitive scripts are used in the model to represent both the actual physical watch (for the user) and visualisation of its internal state (for the designer). These two distinct representations are linked to each other by dependency. By having scripts representing the state of the model, the user who interacts with it freely can observe the behaviour and mechanism inside through stimulus-response patterns.

  The model shows that definitive scripts can be used to implement interactive statecharts (i.e. changing the states of the watch reflects the change of the state in the statechart interactively). The internal state of the watch (observed by the designer) is represented through the statecharts. The choice of the buttons pressed by the user will affect the state of the statecharts and their visualisation.

- Harel's statechart concept [Har87] is a particularly subtle mode of closed-world state representation. The power of definitive representation of the state is illustrated by the fact that the Digital Watch model (comprising 700 definitions) captures the entire state of the statechart together with the complete model of the functionality of the digital watch and the analogue clock. What is more, the true scope for interaction with the definitive model is not represented by the functionality that can be accessed through the interface depicted in Figure 3-17 alone. As will be discussed in more detail in Chapter 5, the functionality of the digital watch as a device is derived by restricting the much richer interaction with the definitive model which is possible via the (d)tkeden input window. This interaction through the input window reflects the modeller's construal (cf. Figure 2-17) of the relationship between the digital watch, analogue clock and statechart which supplies the basis for the model building.

- The Digital Watch model can be viewed as an ADM artefact in which the entities include the components of the statechart in Figure 3-17, viz. *main, power, light, alarm-st* etc. Such an artefact can be derived from an LSD account [ModelWeb] in which these components feature as agents. Under this interpretation, the current state of the statechart represents the set of currently instantiated entities or equivalently the set of currently active agents. Orthogonality in the statechart is then associated with agents acting concurrently. For instance, within the *alive* state of the statechart, the agents *main()*, *light()*, *alarm_st()* and *chime_st()* are active concurrently. Depth in the statechart is similarly associated with the different roles that each agent can play. For instance, the *displays()* agent can play the roles of the agents *time(), date(), update()* etc., where *update()* can itself play the roles of *t-min()*, *min()*, *sec()* etc. We can interpret the statechart not only as specifying rigid patterns of state change that match the machine-like interpretation of an ADM artefact depicted in Figure 3-4, but also as restricting the entity instantiation to a static pattern.

Summary of Section 3.2.4

The MBF model is developed in an exploratory manner without any predefined or well-designed structure. It is incrementally refined and cultivated according to changes in the modeller's perspective and experience. In contrast, the Digital Watch model is conceived with a well-designed structure based on Harel's statechart. The patterns of state-change inside the model are more rigid and structured. Although these two models are developed with fundamentally different orientations towards open and closed development, they are both open and flexible in character. This is because they are constructed using MWDS.