Chapter 6
Case Studies

In accordance with the main principle of EM – knowing-by-doing – the best way to understand the issues discussed in the last two chapters is to undertake practical case studies. In fact, some points of view underlying DEM – such as A-modellers acting as agents to shape agency – and some of the requirements promoting the features of dtkeden – such as synchronous communication, the virtual agent, generic observable, and situated agency – are motivated by practice. In this respect, case studies not only make it possible to experience and explore DEM and dtkeden, but also serve to enrich and expand both in surprising ways.

6.0 Overview

This chapter seeks to exemplify DEM and some features of dtkeden through practical research. Three case studies are provided.

The first case study (6.1) involves modelling a nineteenth-century railway accident. The modelling aims to demonstrate DEM discussed in Chapter 4 (4.2). Several computer models are constructed as artefacts in a distributed modelling environment (that is, dtkeden) in order to shape the agency of agents associated with the accident by means of the successive interactions between A-modellers, who act as these agents, and their models. The concept of pretend play proposed in the last chapter is illustrated by the example of shaping agency that allows two signalmen to change the telegraph needle's position for communication. Also, in the animation of this historic accident, the role of the S-modeller in setting diverse contexts for A-modellers and authorising the accessibility of each agent to observables is illustrated by examples.

Another case study (6.2) illustrates the use of the virtual agent concept described in the last Chapter (5.3). In Section 6.2.1, an ADM translator developed by S. Yung [Yun92] for translating an ADM model to an Eden model is reviewed. A particular problem concerning the Eden models generated by the translator is their readability, because many additional string handling symbols are introduced into the models. A new ADM translator for reengineering this translation has been developed by the present author. The new translator generates Eden models in the virtual agent form. Without the additional symbols, the generated models are much easier to read and maintain.

In addition, Section 6.2.2 provides two examples of adaptable reuse, an important application of the virtual agent concept as discussed in Section 5.3. Within both examples, reusable definitive patterns with a virtual agent are established through the generalisation process (explained in 5.3). These patterns can be particularised to create the required definitive scripts in accordance with the context of the specified virtual agent in run time. Both examples highlight the benefits of using the virtual agent concept to reduce the size of the developed systems and to create definitive patterns for adaptable reuse dynamically, whilst the model is executing.

Section 6.3 illustrates the use of the interaction modes implemented in dtkeden (see Section 5.2). Unlike the two case studies in the previous sections, which make use of the normal interaction mode, the third case study introduced in this section shows the application of the other three interaction modes commonly used for concurrent modelling.

Two systems that have been created by extending pre-existing stand-alone versions in tkeden are given as examples.

6.1 A Railway Accident in the Clayton Tunnel

Table 6-1 describes a railway accident that occurred in the Clayton Tunnel near Brighton in 1861 [Rolt82]. The accident has been studied by using DEM and dtkeden in order to

The Clayton Tunnel Disaster **August 25th 1861**

Three heavy trains leave Brighton for London Victoria on a fine Sunday morning.

They are all scheduled to pass through the Clayton Tunnel---the first railway tunnel to be protected by a telegraph protocol designed to prevent two trains being in the tunnel at once. Elsewhere, safe operation is to be guaranteed by a time interval system, whereby consecutive trains run at least 5 minutes apart. On this occasion, the time intervals between the three trains on their departure from Brighton are 3 and 4 minutes.

There is a signal box at each end of the tunnel. The North Box is operated by **B**rown and the South by **K**illick. K has been working for 24 hours continuously. In his cabin, he has a clock, an alarm bell, a single needle telegraph and a handwheel with which to operate a signal 350 yards down the line. He also has red (stop) and white (go) flags for use in emergency. The telegraph has a dial with three indications: NEUTRAL, OCCUPIED and CLEAR.

When K sends a train into the tunnel, he sends an OCCUPIED signal to B. Before he sends another train, he sends an IS LINE CLEAR? request to B, to which B can respond CLEAR when the next train has emerged from the North end of the tunnel. The dial at one end of the telegraph only displays OCCUPIED or CLEAR when the appropriate key is being pressed at the other---it otherwise displays NEUTRAL.

The distant signal is to be interpreted by a train driver either as *all clear* or as *proceed with caution*. The signal is designed to return to *proceed with caution* as a train passes it, but if this automatic mechanism fails, it rings the alarm in K's cabin.

The accident

When train 1 passed K and entered the tunnel the automatic signal failed to work. The alarm rang in K's cabin. K first sent an OCCUPIED message to B, but then found that train 2 had passed the defective signal before he managed to reset it. K picked up the red flag and displayed it to Scott, the driver of train 2, just as his engine was entering the tunnel. He again sent an OCCUPIED signal to B.

K did not know whether train 1 was still in the tunnel. Nor did he know whether S had seen his red flag. He sent an IS LINE CLEAR? signal to B. At that moment, B saw train 1 emerge from the tunnel, and responded CLEAR. Train 3 was now proceeding with caution towards the tunnel, and K signalled all clear to the driver with his white flag.

But S had seen the red flag. He stopped in the tunnel and cautiously reversed his train to find out what was wrong from K.

Train 3 ran into the rear of Train 2 after travelling 250 yards into the tunnel, propelling Train 2 forwards for 50 yards. The chimney of the engine of Train 3 hit the roof of the tunnel 24 feet above. In all 23 passengers were killed and 176 were seriously injured.

Table 6-1. An account of the Clayton Tunnel railway accident (from [Bey98])

illustrate, and in fact also enrich and expand, DEM as well as the features of dtkeden. This section will focus on illustrating DEM by modelling the accident. The features of dtkeden are then illustrated by other examples shown in the next two sections (Section 6.2 and 6.3).

Modelling the railway accident has involved constructing computer-based artefacts to represent the perspectives of five human agents involved in the accident. These artefacts are also co-ordinated from the point of reference of the S-modeller, that is, an external observer with exceptional state-changing privileges. One significant motivation for building such a model is to gain insight into the individual understandings of the signalmen and drivers concerning their work practices at the time, and to explore how they may have contributed to the accident. The main insight gained concerns the interaction between agents. These agents could include the telegraphs, the alarm, the signal, the signalmen and the drivers. An LSD account for these agents is shown in Appendix 6-A, and Figure 6-1 depicts the distributed modelling environment of modellers. In terms of how the individual understandings of the participants contributed to the accident, this model then focuses on the animation of the accident, and the involvement and exploration of the S-modeller.

In order to capture the individual understanding of each agent, A-modellers can ask questions such as: How did the signalmen communicate with each other via the telegraphs? How big was the red flag? From what distance could the driver see the signal? On the basis of DEM, answers to such questions can be shaped through the interaction between A-modellers by means of the concept of pretend play referred to in section 4.2.

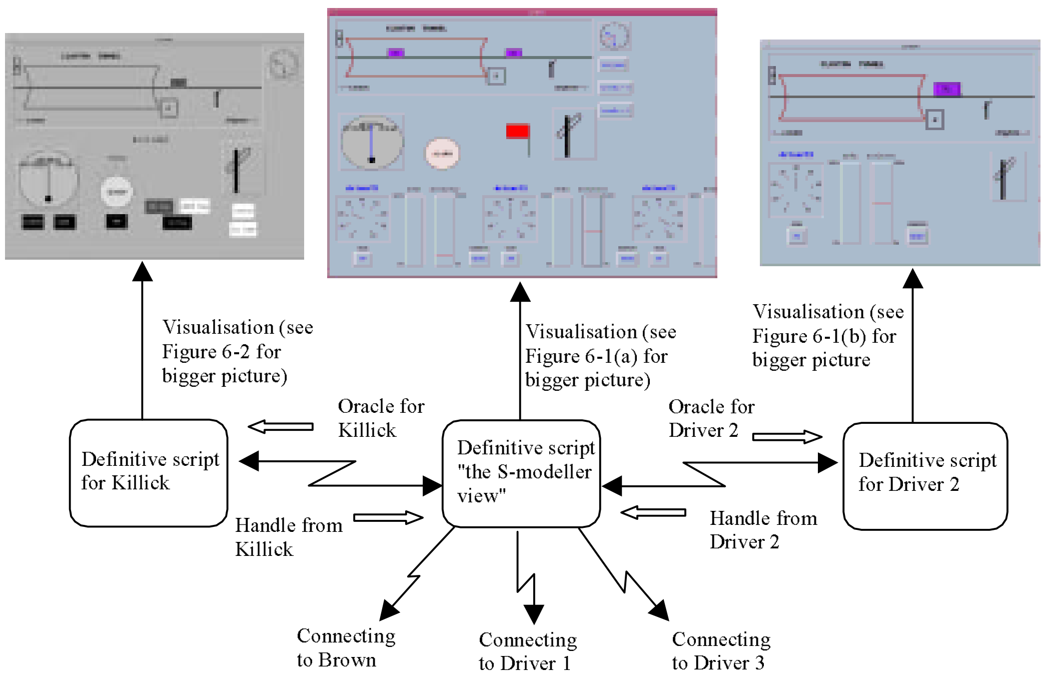

Visualisation (see Figure 6-2 for bigger picture)
Visualisation (see Figure 6-1(a) for bigger picture)
Visualisation (see Figure 6-1(b) for bigger picture
Oracle for Killick
Oracle for Driver 2
Definitive script for Killick
Definitive script "the S-modeller view"
Definitive script for Driver 2
Handle from Killick
Handle from Driver 2
Connecting to Brown
Connecting to Driver 1
Connecting to Driver 3

Figure 6-1. The modelling environment for the railway accident

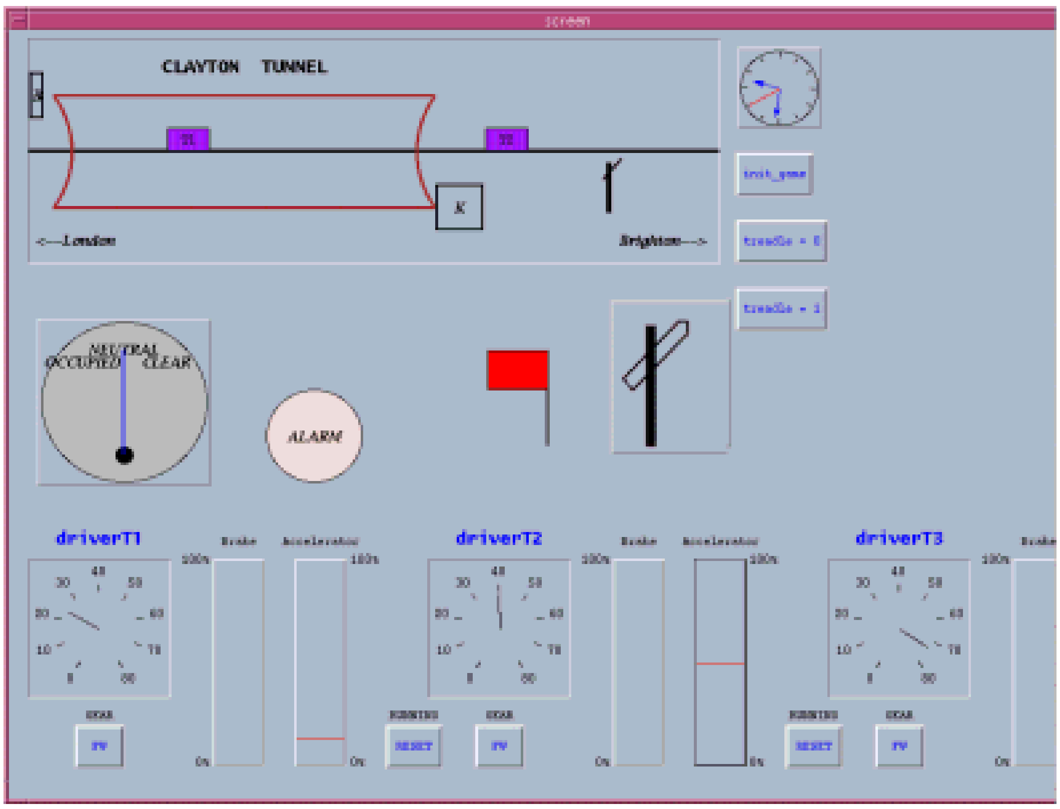


Figure 6-1 (a) A global view of the Clayton Tunnel

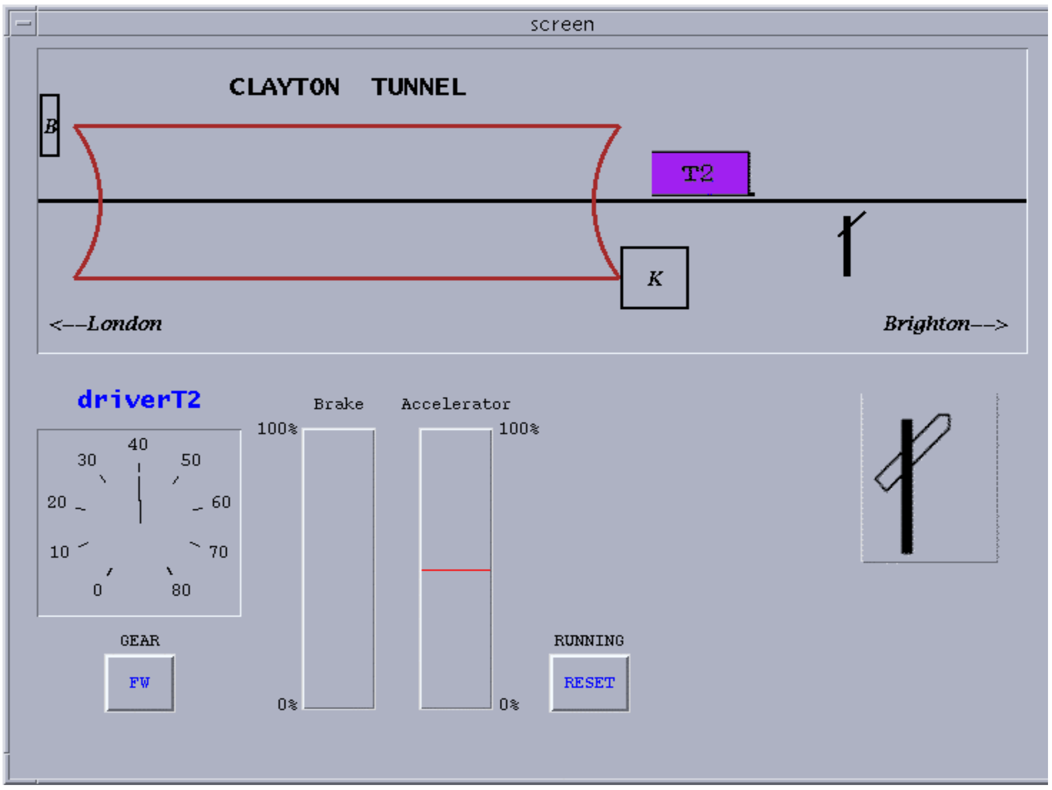


Figure 6-1 (b) The second driver's view of the Clayton Tunnel

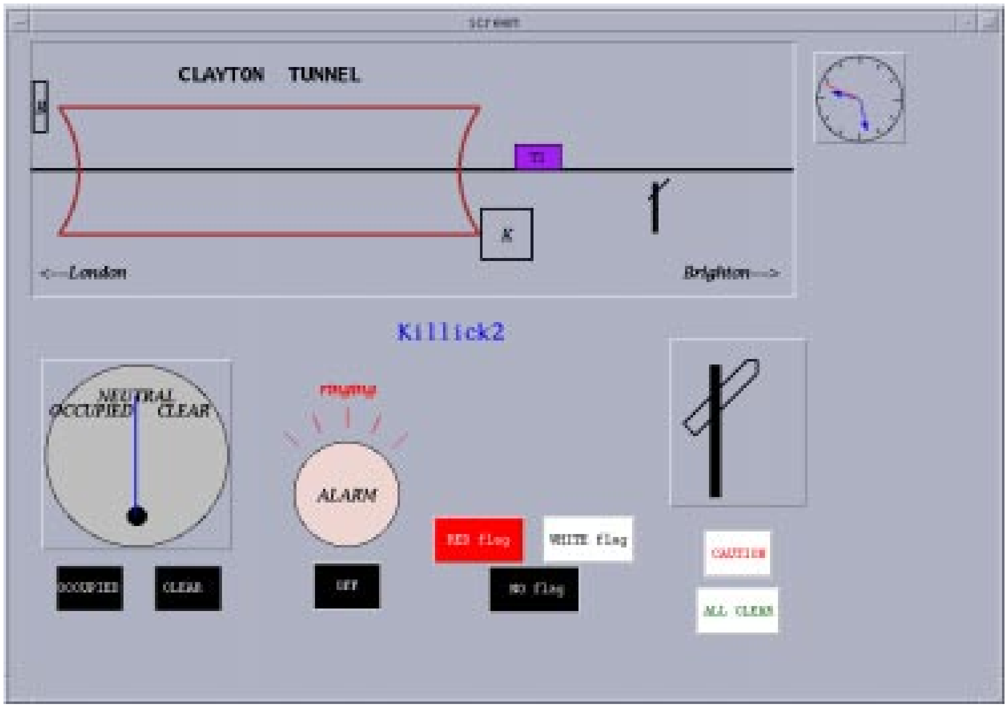


Figure 6-2. A signalman's view of the Clayton Tunnel

By way of illustration, consider how two A-modellers acting in the roles of signalmen can shape the agency involved in telegraph use. In reality, two signalmen, **K**illick and **B**rown, whose location is indicated by the boxes labelled K and B in Figure 6-2 (which represents a view of the Clayton Tunnel from the perspective of the signalman Killick), communicate with each other via the telegraph shown in the bottom-left corner of the figure. To represent this agency, two A-modellers define their private views as follows:

Signalman Killick:

```
 NeedlePos  is  clicked;
 clicked is  (click_clear)?(1):((click_neutral)?(0):((click_occupied)?(-1)));
 clear_clicked is Killick_click_clear;
 neutral_clicked is Killick_click_neutral;
 occupied_clicked is Killick_click_occupied;
```

Signalman Brown:

```
NeedlePosition  is  which_clicked;
which_clicked is  (clear_clicked)?(1):((neutral_clicked)?(0):((occupied_clicked)?(-1)));
click_clear is Brown_click_clear;
click_neutral is Brown_click_neutral;
click_occupied is Brown_click_occupied;
```

Definitions of this nature specify computational artefacts to represent the personal agent perspectives. Different naming conventions for observables are used to reflect the independence of the agent's observation. (It should be noted that, for simplicity, the definitions concerned with visualisation are omitted.) The order in which the various definitions are introduced is not as important as in traditional programming because dtkeden automatically maintains dependencies amongst observables. For example, in the case of Brown, the value of the observable 'needlePos' will be changed whenever the value of the observable 'clicked' is changed. Further description is needed to express the way in which the signalmen inform each other about the state of the tunnel. This could be as follows:

```
Signalman Killick:
Func send1: Killick_click_clear{
  if (click_clear)
    sendAgent("Brown", "Killick_click_clear=TRUE;");
  else
    sendAgent("Brown", "Killick_click_clear=FALSE;");
}
Func send2: Killick_click_neutral{
  if (click_neutral)
    sendAgent("Brown", "Killick_click_neutral=TRUE;");
  else
    sendAgent("Brown", "Killick_click_neutral=FALSE;");
}
Func send3: Killick_click_occupied{
   If (click_occupied)
    sendAgent("Brown", "Killick_click_occupied=TRUE;");
   else
    sendAgent("Brown", "Killick_click_occupied=FALSE;");
}
```

```
Signalman Brown:
Func sendA: Brown_click_clear{
  if (click_clear)
    sendAgent("Killick", "Brown_click_clear=TRUE;");
  else
    sendAgent("Killick", "Brown_click_clear=FALSE;");
}
Func sendB: Brown_click_neutral{
  if (click_neutral)
    sendAgent("Killick", "Brown_click_neutral=TRUE;");
  else
    sendAgent("Killick", "Brown_click_neutral=FALSE;");
}
Func sendC: Brown_click_occupied{
  if (click_occupied)
    sendAgent("Killick", "Brown_click_occupied=TRUE;");
   else
    sendAgent("Killick", "Brown_click_occupied=FALSE;");
}
```

Although the above mechanisms send definitions from one signalman to the other, this does not of itself establish useful communication. To this end, there must also be a dependency between the definitions received by a signalman acted by an A-modeller and the private definitions within its own computational artefact. When redefinitions such as the following are added to their computer-based models, the agency of changing the telegraph's states for communication is established:

```
Signalman Killick:
  clear_clicked is Killick_click_clear or Brown_click_clear;
  neutral_clicked is Killick_click_neutral or Brown_click_neutral;
  occupied_clicked is Killick_click_occupied or Brown_click_occupied;

Signalman Brown:
  click_clear is Brown_click_clear or Killick_click_clear;
  click_neutral is Brown_click_neutral or Killick_click_neutral;
  click_occupied is Brown_click_occupied or Killick_click_occupied;
```

This example illustrates how A-modellers on the basis of DEM act as agents to shape the agency of agents through the interaction with each other (that is, in the form of pretend play proposed in Section 4.2). In the early stage of modelling this railway accident, the agency that is illustrated above took the following form: a signalman presses a key on his telegraph to set both needles' positions to indicate the state of the tunnel, such as 'occupied' and 'clear', and the other signalman resets the positions. After experiments with the computer model, this agency was revised so as to take the following form[1]: a signalman holds the key down to keep the needles at a position for a while and then releases the key to return the needles to the 'neutral' position. This evolution of agency in this model highlights the significance of shaping agency through the interaction between agents acted by A-modellers. For A-modellers, the concept of pretend play can help not only to shape the agency of agents but also to improve their understanding of these agencies through enabling them to gain experience of the state change caused by their interaction from their computer-based models.

For the purpose of animating the accident, the S-modeller, who can exercise super agency in providing A-modellers with a particular context, is involved. The S-modeller, for example, can make the signal definitely fail to work by refining the observable

[1] The model has been constructed in the absence of explicit information about how signalmen at the Clayton Tunnel communicated with each other by using two telegraphs in 1861. However, the revised scenario for communication reflects our knowledge of the technology of the time (for example, the fact that electric current had to be generated by a manually operated dynamo) and allows us to interpret the interactions between the signalmen Killick and Brown, as recorded by the accident enquiry (in particular, the information that Killick sent an IS LINE CLEAR? signal to Brown and Brown replied CLEAR).

treadle_reliability equal to 0 (that is, treadle_reliability = 0)[2] in order to explore the consequent interaction between A-modellers. Exploration through 'what if' experiments provides the S-modeller with contextual resources to gain his/her insight into the interaction between agents. The S-modeller can undertake many 'what if' experiments to explore the reasons for the accident: for example, slowing down the speed of train 2, increasing the time interval between trains, enlarging the distance between Killick and the signal, and so on. It is to be expected that the S-modeller can broaden his/her insight into the accident as a result of this arbitrary exploration. This open-ended, situated modelling, that provides the S-modeller with more contextual resources, is hard to achieve by traditional modelling methods, where the allowed exploration has to be anticipated.

The potential usefulness of the computer model to an accident investigator can be illustrated by a number of 'what if' experiments. In this way, the author was able to identify scenarios to show the responsibility for the accident cannot be pinned on any one agent. For example, let us consider the case in which driver Scott sees the red flag and just stops his train in the tunnel rather than reversing it. Since Killick has seen the 'CLEAR' message from Brown, he decides to wave his white flag to inform train 3 to enter the tunnel. When train 3 enters the tunnel, the driver does not expect another train to be there (because there is no accessibility to observe train 2's position) and there is no time to stop his train before crashing into train 2. In that case, the accident occurs even though Scott follows his protocol to the letter. As a second example, consider the scenario in which Killick questions the meaning of the 'CLEAR' message and decides to stop train 3 from entering the tunnel. Since the driver of train 2 (Scott) has seen the red flag, he decides to reverse his train in order to discover what is wrong. When he is reversing the train, he is not aware that another train has stopped just in front of the entrance of the tunnel. Though Killick can see the positions of both trains, he cannot stop them, since he

[2] For reasons of convenience in animating the accident, the buttons used to set the signal as definitely working or not working are implemented in the S-modeller's model as shown in Figure 6-1(a)

has no direct control over their movements. This helps to clarify Killick's role in the accident. Analyses of this sort can show that no single agent should be blamed for the accident. It is more accurate to conclude that the accident is due to the collaborative interaction between all the agents.

Another important task performed by the S-modellers is to determine the accessibility of observables by each agent (see Section 5.2 for a more detailed explanation). For example, the following LSD account is given by the S-modeller to describe how certain observables can be accessed by the two signalman agents.

```
%lsd
agent Killick
oracle _FLAG_showing, showingFlagColour, _FLAG_flagpole_pos_x, _FLAG_goLeft
oracle _ALARM_ringing , _ALARM_flash, _TELEGRAPH_needle_pos, signalSign
oracle _CLOCK_hour, _CLOCK_min, _CLOCK_sec

handle _FLAG_showing, showingFlagColour, _FLAG_flagpole_pos_x, _FLAG_goLeft
handle _TELEGRAPH_needle_pos, signalSign, _ALARM_ringing

agent Brown
oracle _TELEGRAPH_needle_pos
handle _TELEGRAPH_needle_pos
```

Not all observables can be described in advance. For example, the observable T2*_TRAIN_train_pos_x* representing the position of train T2 occurs only if train T2 starts to move. Also, the accessibility of each agent to this observable is not persistent. In fact, signalman Killick has access to this observable when the train is in a position where it can be seen by him. In particular, Killick cannot access this observable when the train enters the tunnel. An Eden action to implement such a privilege is given as follows (only a part of the action is shown here, due to limitations of space):

```
proc changeTrainAgency : _TRAIN_train_pos_x, driverSeeFlagPos, KillickSeeTrainPos {
    auto  trainPos;
    trainPos="_"//str(eval(~trainDriver))//"_TRAIN_train_pos_x";
    if (_TRAIN_train_pos_x - ~_SITE_Killick_pos <= driverSeeFlagPos &&
        _TRAIN_train_pos_x >= ~_SITE_tunnel_r_pos && !driver_see_flag) {
       addAgency("Killick", "oracle", trainPos);
       Killick_see_train = TRUE;
    }
    if ((_TRAIN_train_pos_x >= ~_SITE_Killick_pos + KillickSeeTrainPos ||
        _TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_tunnel_r_pos) &&
       Killick_see_train) {
       removeAgency("Killick", "oracle", trainPos);
       sendClient("Killick", trainPos // " = 99999;\n");
       Killick_see_train = FALSE;
    }
}
```

In this Eden action, changes to the train's position and to the distances at which objects become visible serve as triggers to redefine Killick's privilege to observe the train. The model assumes that Killick's flag will be visible to the train driver precisely when the train is visible to Killick.

The railway accident case study[3] illustrates very well not only the points of view underlying DEM but also the key features of dtkeden for supporting distributed modelling: for example, synchronous communication (among Killick, Brown and each driver), and reusable definitive patterns (see the way in which trains are specified in Appendix 6-B for example). The next two sections illustrate these features in more detail.

[3] More details of the accident animation are available on the website http://dcs.warwick.ac.uk//modelling/railway developed by S. Maad.

6.2 The Application of the Virtual Agent Concept

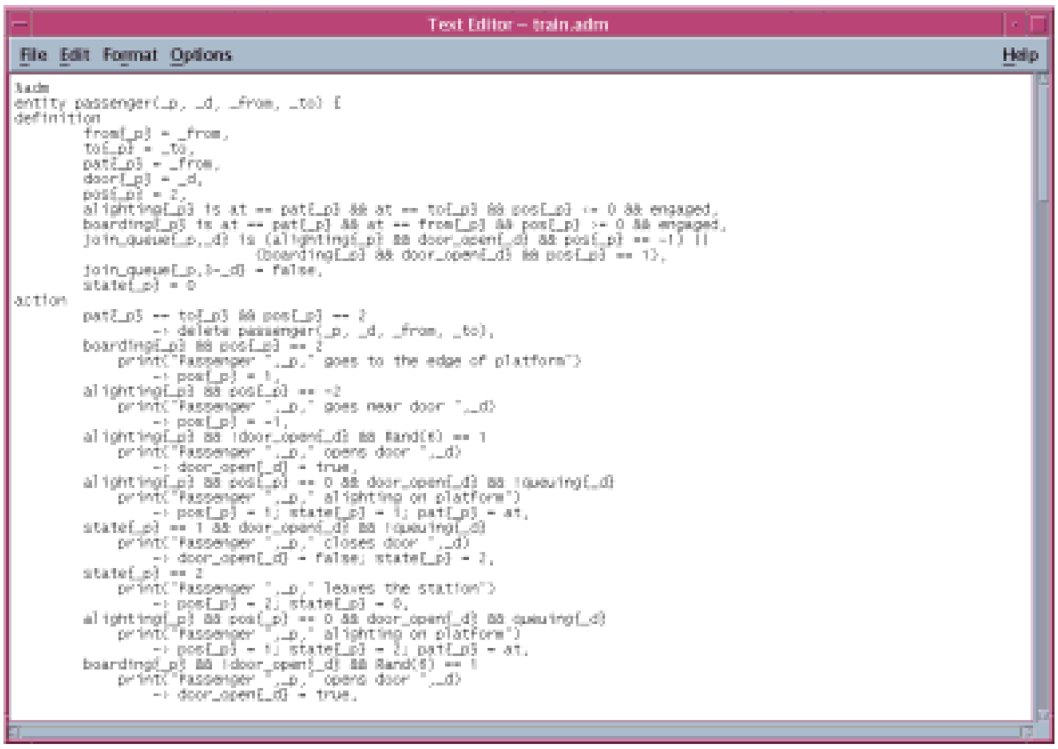
One of the most important features of dtkeden is the virtual agent concept. As described in section 5.3, the use of the virtual agent concept enables a modeller not only to localise a definitive script in accordance with a specific context but also to generate scripts for reuse. This section gives examples of using the virtual agent concept.

6.2.1 Reengineering ADM

As described earlier (2.3.1), the ADM is an abstract machine which has been developed to give operational meaning to the characters of parallel state-change and openness in an LSD account [Sla90]. An LSD account intended for describing systems at a higher-level abstraction is non-executable. In order to create a computer-based model to animate the description in an LSD account, the development of an abstract machine such as the ADM is required. With the ADM, the modeller can animate the behaviour described by an LSD account and intervene in this animation by (re)definition through definitive programming.

The task of translating an LSD account into an ADM model is so far performed manually. The account of each LSD agent is presented in the form of an entity, which includes a definition section and an action section. A general rule for this translation is that variables owned by the agent are put into the definition section and the description of *protocol* is put into the other section in the form of action (For precise details of this translation, see [Sla90]).

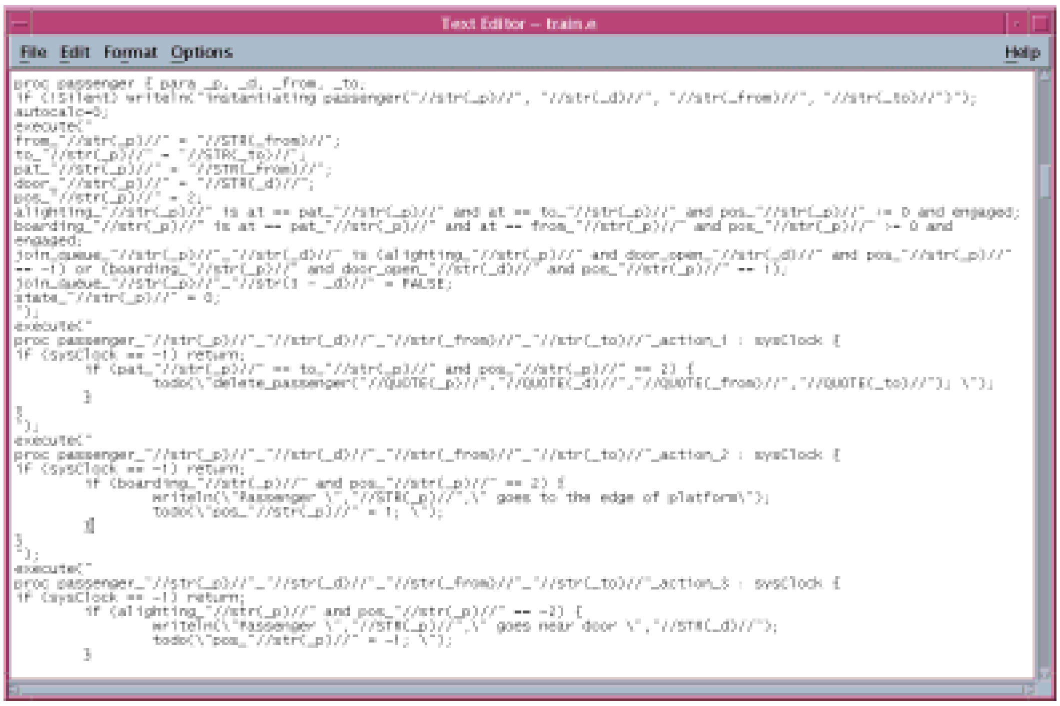
Though Slade's implementation of the ADM is a successful proof-of-concept tool to demonstrate how an animation can be created from an LSD account, its applications are limited. This is because, in this implementation, the ADM only accepts the integer data type, and has no support for visualisation. Hence, for most applications, ADM models are translated into Eden models rather than interactively interpreted. The translation can be divided into two parts: system and application. The system part embedded in each translated Eden model deals with the animation of parallel state-change in the ADM, and the application part is generated by translating entities in the ADM into Eden notation (such as definitions and actions).



```
%adm
entity passenger(_p, _d, _from, _to) {
definition
        from[_p] = _from,
        to[_p] = _to,
        pat[_p] = _from,
        door[_p] = _d,
        pos[_p] = 2,
        alighting[_p] is at == pat[_p] && at == to[_p] && pos[_p] <= 0 && engaged,
        boarding[_p] is at == pat[_p] && at == from[_p] && pos[_p] >= 0 && engaged,
        join_queue[_p,_d] is (alighting[_p] && door_open[_d] && pos[_p] == -1) ||
                             (boarding[_p] && door_open[_d] && pos[_p] == 1),
        join_queue[_p,!=_d] = false,
        state[_p] = 0
action
        pat[_p] == to[_p] && pos[_p] == 2
                -> delete passenger(_p, _d, _from, _to),
        boarding[_p] && pos[_p] == 2
            print("Passenger ",_p," goes to the edge of platform")
                -> pos[_p] = 1,
        alighting[_p] && pos[_p] == -2
            print("Passenger ",_p," goes near door ",_d)
                -> pos[_p] = -1,
        alighting[_p] && !door_open[_d] && Rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open[_d] = true,
        alighting[_p] && pos[_p] == 0 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," alighting on platform")
                -> pos[_p] = 1; state[_p] = 1; pat[_p] = at,
        state[_p] == 1 && door_open[_d] && !queuing[_d]
            print("Passenger ",_p," closes door ",_d)
                -> door_open[_d] = false; state[_p] = 2,
        state[_p] == 2
            print("Passenger ",_p," leaves the station")
                -> pos[_p] = 2; state[_p] = 0,
        alighting[_p] && pos[_p] == 0 && door_open[_d] && queuing[_d]
            print("Passenger ",_p," alighting on platform")
                -> pos[_p] = 1; state[_p] = 2; pat[_p] = at,
        boarding[_p] && !door_open[_d] && Rand(6) == 1
            print("Passenger ",_p," opens door ",_d)
                -> door_open[_d] = true,
```

Figure 6-3. A snapshot of the entity *passenger* in ADM

A translator named adm has been developed to support the translation [Yun92]. With adm, in addition to the system part, each entity description is translated into a procedure (with the same name as the entity) in which the textual forms of the definition and the action sections of the entity are treated as parameters to an Eden built-in function called *execute*. Once the procedure is called, Eden definitions and actions to represent the original ADM entity are generated. This same procedure can be used when the model is running (with specified parameters, if appropriate) in order to instantiate Eden definitions and actions to reflect the (specified) context of the entity. Figures 6-3 and 6-4 illustrate the translation by adm in which an entity called *passenger* is translated into a procedure

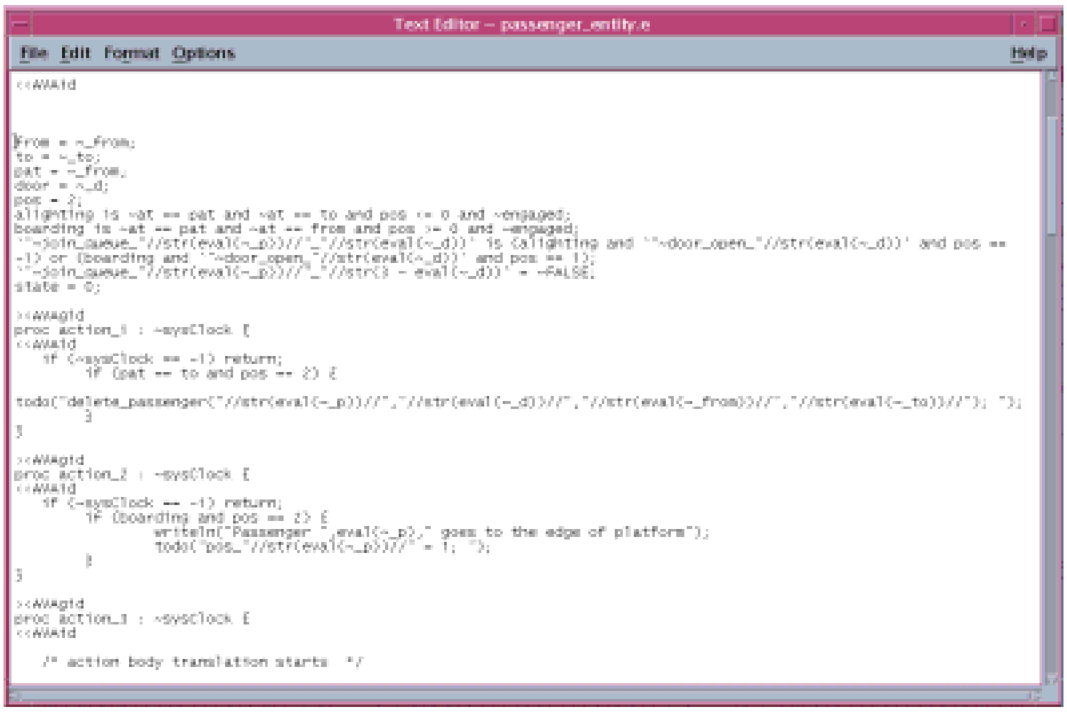


```
proc passenger { para _p, _d, _from, _to;
if (!Silent) writeln("instantiating passenger("//str(_p)//", "//str(_d)//", "//str(_from)//", "//str(_to)//")");
autocalc=0;
execute("
from_"//str(_p)//" = "//STR(_from)//";
to_"//str(_p)//" = "//STR(_to)//";
pat_"//str(_p)//" = "//STR(_from)//";
door_"//str(_p)//" = "//STR(_d)//";
pos_"//str(_p)//" = 2;
alighting_"//str(_p)//" is at == pat_"//str(_p)//" and at == to_"//str(_p)//" and pos_"//str(_p)//" != 0 and engaged;
boarding_"//str(_p)//" is at == pat_"//str(_p)//" and at == from_"//str(_p)//" and pos_"//str(_p)//" >= 0 and
engaged;
join_queue_"//str(_p)//"_"//str(_d)//" is (alighting_"//str(_p)//" and door_open_"//str(_d)//" and pos_"//str(_p)//"
== -1) or (boarding_"//str(_p)//" and door_open_"//str(_d)//" and pos_"//str(_p)//" == 1);
join_queue_"//str(_p)//"_"//str(1 - _d)//" = FALSE;
state_"//str(_p)//" = 0;
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_1 : sysClock {
if (sysClock == -1) return;
        if (pat_"//str(_p)//" == to_"//str(_p)//" and pos_"//str(_p)//" == 2) {
                todo(\"delete_passenger("//QUOTE(_p)//","//QUOTE(_d)//","//QUOTE(_from)//","//QUOTE(_to)//"); \");
        }
}
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_2 : sysClock {
if (sysClock == -1) return;
        if (boarding_"//str(_p)//" and pos_"//str(_p)//" == 2) {
                writeln(\"Passenger \","//STR(_p)//",\" goes to the edge of platform\");
                todo(\"pos_"//str(_p)//" = 1; \");
        }
}
");
execute("
proc passenger_"//str(_p)//"_"//str(_d)//"_"//str(_from)//"_"//str(_to)//"_action_3 : sysClock {
if (sysClock == -1) return;
        if (alighting_"//str(_p)//" and pos_"//str(_p)//" == -2) {
                writeln(\"Passenger \","//STR(_p)//",\" goes near door \","//STR(_d)//");
                todo(\"pos_"//str(_p)//" = -1; \");
        }
```

Figure 6-4. A snapshot of the Eden scripts generated for the entity *passenger* by the original version of the ADM translator

(also called *passenger*). As shown in Figure 6-4, the Eden scripts generated by the translation are difficult to read. This is because additional string handling symbols for the parameters of the function *execute* are inserted. This problem can be almost entirely eliminated by re-engineering this translation.

The use of the virtual agent concept, which can repeatedly generate similar definitive scripts from a definitive pattern (explained in Section 5.3), is able to serve the purpose of instantiation in adm. This has motivated the author to develop a new translator named adm3. In adm3, each entity is translated into a definitive pattern with an unspecified virtual agent, as discussed in section 5.3. The definitive patterns stored in individual files can then be reused to generate similar scripts according to the contexts of their specified virtual agents. The function *execute* is no longer used, so that additional string handling symbols are almost unnecessary. In order to make it possible to reuse each definitive pattern describing an entity, adm3, like adm, generates a procedure with the



```
<<AVAid


from = ~_from;
to = ~_to;
pat = ~_from;
door = ~_d;
pos = 2;
alighting is ~at == pat and ~at == to and pos <= 0 and ~engaged;
boarding is ~at == pat and ~at == from and pos >= 0 and ~engaged;
`"~join_queue_"//str(eval(~_p))//"_"//str(eval(~_d))` is (alighting and `"~door_open_"//str(eval(~_d))` and pos ==
-1) or (boarding and `"~door_open_"//str(eval(~_d))` and pos == 1);
`"~join_queue_"//str(eval(~_p))//"_"//str(3 - eval(~_d))` = ~FALSE;
state = 0;

>>AVAgid
proc action_1 : ~sysClock {
<<AVAid
   if (~sysClock == -1) return;
        if (pat == to and pos == 2) {

todo("delete_passenger("//str(eval(~_p))//","//str(eval(~_d))//","//str(eval(~_from))//","//str(eval(~_to))//"); ");
        }
}

>>AVAgid
proc action_2 : ~sysClock {
<<AVAid
   if (~sysClock == -1) return;
        if (boarding and pos == 2) {
                writeln("Passenger ",eval(~_p)," goes to the edge of platform");
                todo("pos_"//str(eval(~_p))//" = 1; ");
        }
}

>>AVAgid
proc action_3 : ~sysClock {
<<AVAid

   /* action body translation starts  */
```

Figure 6-5. The Eden scripts generated for the entity *passenger* by the author's revised version of the ADM translator

name of this entity. This procedure is much simpler than that generated by adm, since most of its content is extracted and stored into a file as a reusable definitive pattern. In effect, this procedure serves to perform the process of particularising a reusable definitive pattern as discussed in Section 5.3. It specifies the context of the virtual agent and the parameters, if any, and then includes the file of this definitive pattern in order to reuse the definitive pattern in the current context. Figure 6-5 shows the definitive pattern generated by adm3 for the entity *passenger* in Figure 6-3. Compared with the Figure 6-4, it is clear that the generated Eden model, in which there are no additional string handling symbols, is easier to read and maintain. In respect of the translation of the system part, adm3 is exactly the same as adm.

6.2.2 Other Examples

In addition to the application of reengineering the ADM translator, the use of the virtual agent concept for adaptable reuse has been illustrated in several projects. By reusing adaptable definitive patterns, these projects not only succeed in simplifying the programmer's task but also benefit from reducing model size and generating structured definitive scripts. Two examples are given here:

- The classroom project[4]

This project seeks to develop an animation system to model the interactive behaviours of pupils and the teacher in a classroom. Since the variables associated with a pupil, such as those for showing the face of a pupil, must be defined for each specific pupil in response to the observed world of the modeller, the size of the developed system (for 6 pupils) is large (about 165K bytes in total). This leads to time-consuming inconvenience and difficulty in maintaining the system: for example, the need to change some observables' descriptions and to add more pupil icons into the system on-the-fly. In fact, examining the system carefully, it is found that many chunks of scripts have a high degree of similarity: for example, the description of each pupil's behaviour, the Scout windows for showing pupil icons and measuring the personal characteristics of pupils, and the description of drawing each pupil's face in DoNaLD. These similarities point to the use of the virtual agent concept for reducing size and maintenance load. Corresponding to the modeller's practical experience, GOs (that is, generic observables) and their definitive patterns, for example, *girlface*, *boyface*, *pupil-icon*, can be generalised as reusable patterns. These patterns can then be particularised to create the needed definitive scripts on-the-fly. The generated scripts are adaptable in accordance with their

[4] The project was originally developed by a third-year student in the author's department. To date, its modification by using the virtual agent concept has been developed collaboratively by the present author and another PhD student S. Rasmequan.

specific contexts. For example, the description of the position of each pupil's icon can be modified in response to its real position on the displayed screen.

By exploiting the virtual agent concept, the size of the system has been reduced by 40% (to about 100K bytes). More significantly, these definitive patterns become reusable components and can be conveniently reused with optimal modification on-the-fly. Neither white-box nor black-box reuse provides this feature of adaptable reuse when the model is running, since both kinds of reusable component are well-defined in advance and cannot be modified on-the-fly. Figure 6-6 shows that the system has been extended to 12 pupils just by reusing definitive patterns without additional description.

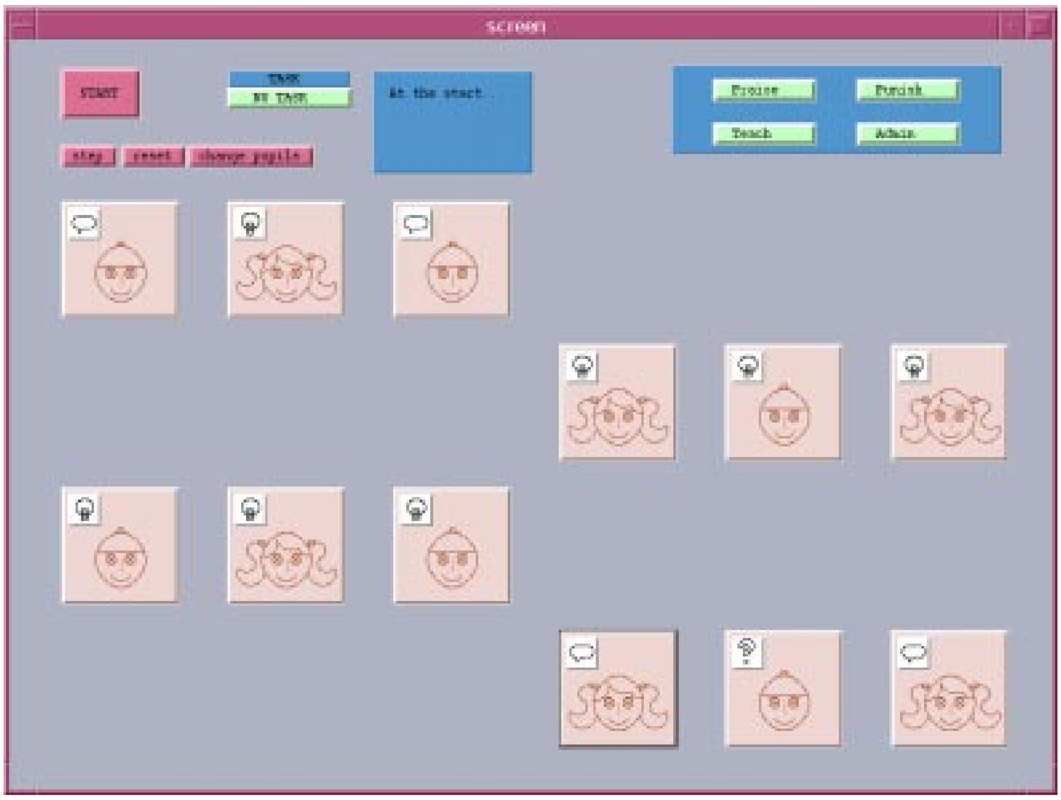
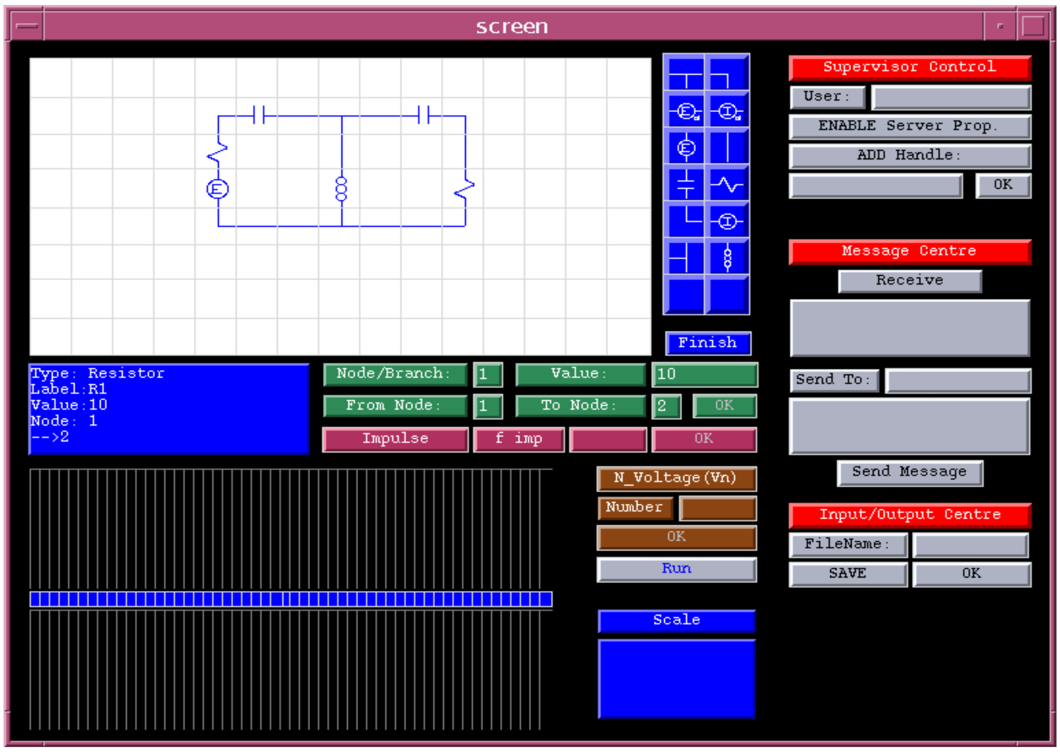


Figure 6-6. The application of reusable definitive patterns in the classroom simulation system

- The virtual electrical laboratory project[5]

This project uses EM to develop a distributed electrical laboratory system for educational purposes. Within the dtkeden environment, the scenarios of the system can be considered as follows: the teacher at the S-node draws up a circuit diagram and sends it to students at A-nodes, and each student interacts with the received diagram by changing its components and their values for learning purposes.



```
Type: Resistor
Label:R1
Value:10
Node: 1
-->2
```

Node/Branch: 1 Value: 10
From Node: 1 To Node: 2 OK

Figure 6-7. A snapshot of virtual electronic laboratory

In order to support the frequent interaction between the teacher and students, and their computer models, a large number of icons are used. For example, a circuit diagram can be drawn by selecting a symbol from an icon bar and pasting the selected symbol into an icon of the workspace consisting of another 120 icons with empty contents. In fact, as shown in Figure 6-7, more than 200 icons are used for the interface to support the

[5] This MSc research project was jointly developed by H. P. D'ornellas [Dor98] and C. R. Sheth [She98].

interaction and display graphic data. Each icon is specified by defining a Scout window to include a DoNaLD picture. As can be imagined, a heavy load of modelling is inevitable for creating and maintaining these icons in tkeden; however, the load can be relieved by using the virtual agent concept. For example, a GO named 'agent.p4' provides a definitive pattern which can be reused for describing icons in the icon bar (more details are given in [She98]).

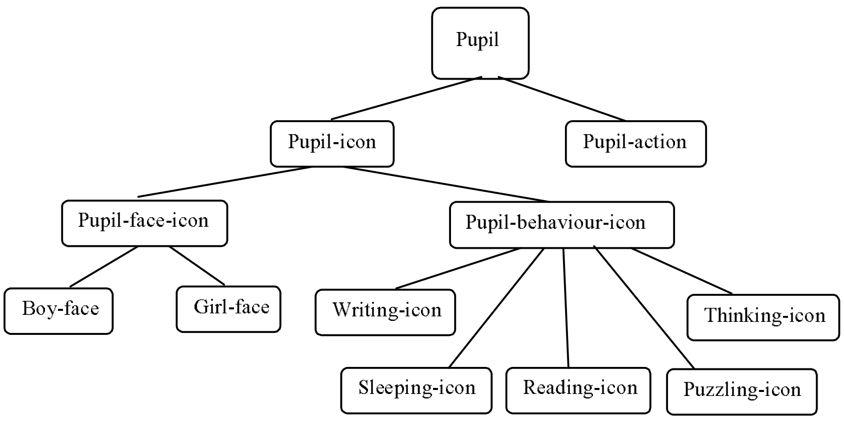


Figure 6-8. The partial hierarchical structure of the modified classroom simulation system

Both the above projects demonstrate the advantages of applying the virtual agent concept to software reuse, in particular for adaptable reuse when the model is running, as discussed in Section 5.3. More significantly, it is found that the creation of definitive patterns facilitates the construction of the hierarchical structure of the developed system. Where definitive programming is used, as in tkeden, it is typically not necessary for the modeller to address issues associated with the structure of models. However, a structure for dtkeden models can be explicitly introduced when reusable definitive patterns in dtkeden are subtly devised. For example, Figure 6-8 illustrates the partial structure of the classroom simulation system (the modified version). It should be noted that such an application of definitive patterns need not be regarded as imposing a fixed structure on the system being developed, due to the adaptability of these patterns.

6.3 Examples of Interaction Modes

There are four interaction modes in dtkeden. The examples in the previous sections are typically in the normal mode. This section includes another two examples developed by the author to illustrate different modes.

- A two-player OXO game

This system was originally developed to model a generic OXO game in the tkeden environment, that is, in a stand-alone environment (details can be found in [Nes97]). The only user of the system is the modeller, who, being the superagent, is empowered to access and change all variables of the computer-based model. When the system is extended to a distributed environment, there is no longer a single super agent.

In the extended OXO model, more modellers are involved: two for players X and O at A-nodes and one for the umpire (associated with 'the S-modeller view') at the S-node. Through a communication network, a player X can send a definitive script, for example a definition s1 = x (describing a cross is placed to position s1), to interact with (e.g. to play OXO with) another player O[6]. Due to the star-type logical network configuration described earlier (Section 5.1), the script is first automatically directed to the S-node. If the interaction mode of the S-node is broadcast mode, the script will be accepted so as to affect the computer model at the S-node, and will concurrently be sent to another player O, leading to the change of the visualisation of player O's computer

[6] Although a player can also input a definitive script without sending it to the umpire, in order to interact with his/her own computer model, the interaction will be regarded as a stand-alone modelling in EM for individual cognition. This is not considered in this section.

model. Figure 6-9 shows the interaction between the three models in this form of broadcasting.

If the interaction mode is declared as interference mode, scripts that are sent to the S-node will be displayed in the umpire's input window pending further action from the umpire (as shown in Figure 6-10). At this point, the umpire can interfere in the interaction between two players, for example by changing the definition s1 = x (received from player X) to s1 = o (signifying the placing of a nought in position s1) and sending it to the player O. As a result, a surprising inconsistency occurs in the players' computer models. The unexpected contexts that arise from interference at the S-node can enrich the understanding of all the modellers involved in the system.

- A monitoring system for an educational game

The jugs model in tkeden implements a simple educational game intended to teach pupils elementary number theory (see [Bey89+, BS98] for details). A dtkeden system has been developed by extending the stand-alone version of the jugs model in tkeden. This system allows a teacher (sited at the S-node) to monitor the progress of several pupils who are independently playing jugs. The interaction mode at the S-node for this system is set to the private mode so as to establish private channels to individual pupils for monitoring their playing context. In this way, each interaction performed by a particular pupil who is playing jugs on his/her computer model is propagated to the model at the S-node and only affects the part of the teacher's model which corresponds to that pupil's context. Figure 6-11 illustrates how different contexts, corresponding to the models of different pupils, can be shown in the same model at the S-node, though these models essentially have the same observables and dependency for each pupil. This system demonstrates the archetypal usage of the virtual agent concept, viz. to localise definitive scripts in accordance with their contexts. This concept of localisation gives rise to the concept of adaptable reuse.

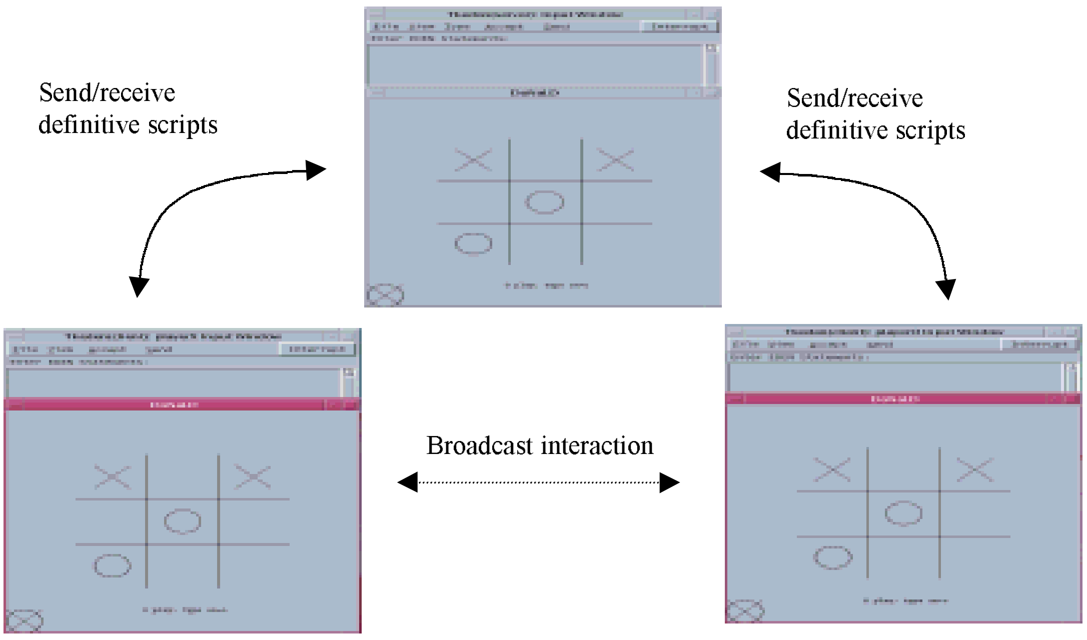


Figure 6-9. Interaction in the broadcast mode

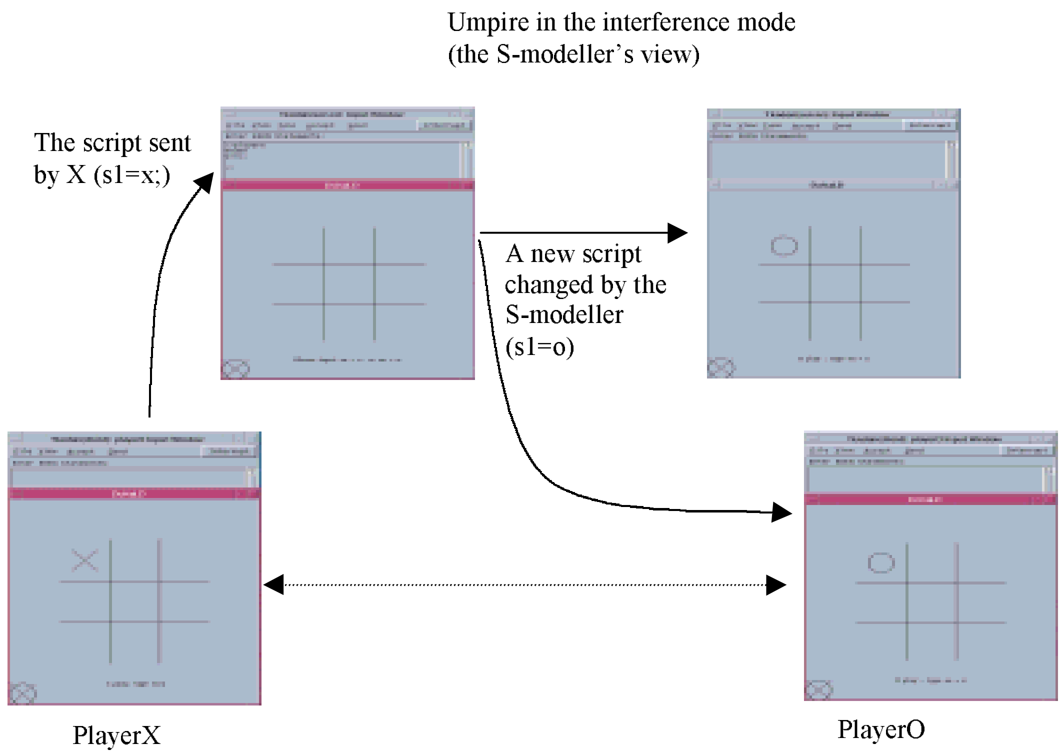
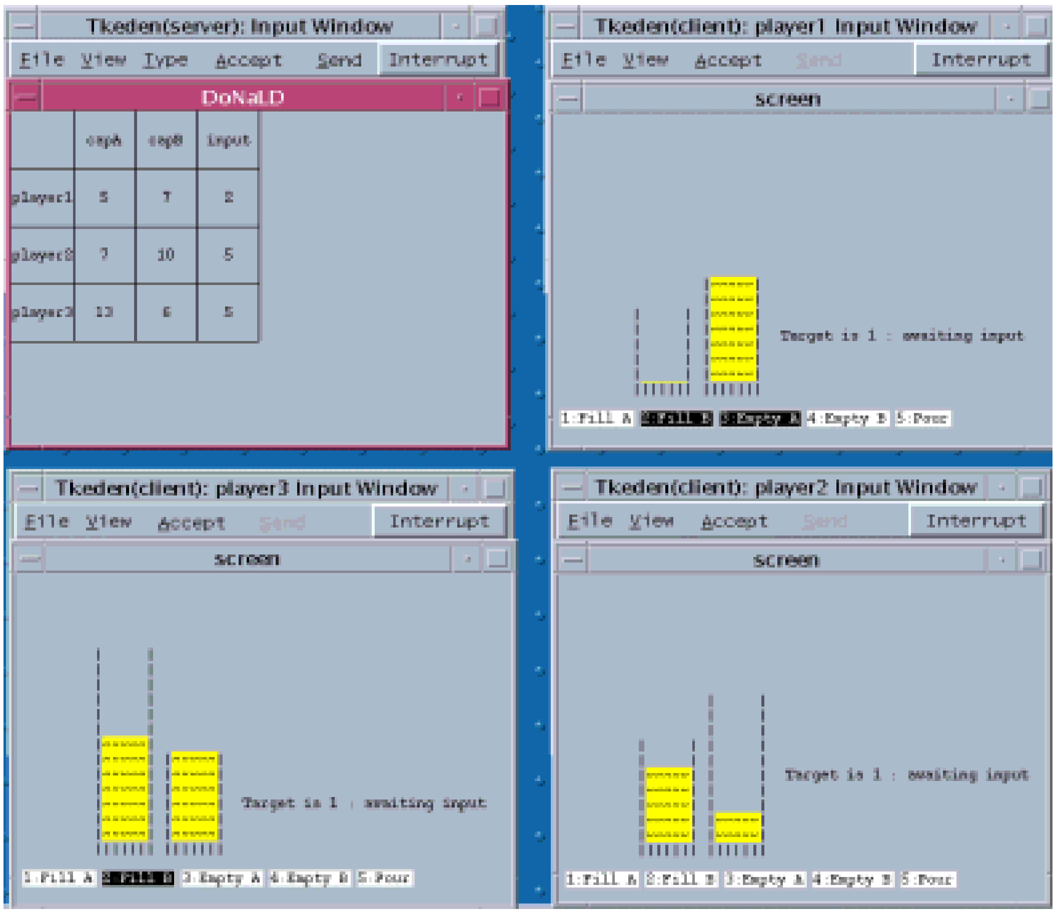


Figure 6-10. Interaction in the interference mode


Tkeden(server): Input Window
File View Type Accept Send Interrupt
DoNaLD
Tkeden(client): player1 Input Window
File View Accept Send Interrupt
screen
1:Fill A 2:Fill B 3:Empty A 4:Empty B 5:Pour
Tkeden(client): player3 Input Window
File View Accept Send Interrupt
screen
1:Fill A 2:Fill B 3:Empty A 4:Empty B 5:Pour
Tkeden(client): player2 Input Window
File View Accept Send Interrupt
screen
1:Fill A 2:Fill B 3:Empty A 4:Empty B 5:Pour

	capA	capB	input
player1	5	7	2
player2	7	10	5
player3	13	6	5

Target is 1 : awaiting input

Target is 1 : awaiting input

Target is 1 : awaiting input

Figure 6-11. Different contexts of a jugs game in the private mode

Appendix 6-A: An LSD Account for the Railway Accident

This appendix presents an LSD account of the agents involved in the Clayton Tunnel railway accident. The account is subject to change as more knowledge emerges from the modelling. In addition, for the sake of convenience, each of the key positions of a train in this account is represented by a simple letter. These symbols and their meaning are explained after the LSD account.

```
agent Killick {
state
    set_alarm_off
    set_signal
    showing_flag
  oracle
    clock_time
    train_position(i)        /* get this agency when he can see the train i, but lost this agency when
                               the train i enters the tunnel. */
    telegraph_needle_position
    alarm_ringing
  handle
    set_needle_position       /* set telegraph_needle_position to OCCUPIED(1) */
    set_signal              /* set signal_sign to ALL_CLEAR(0) or CAUTION(1) */
    set_alarm           /* we assume alarm will keep ringing until Killick reset it */
    showingFlagColour
  protocol
    /* when telegraph needle position is in CLEAR, set it to NEUTRAL
       and reset signal to ALL_CLEAR */
   telegraph_needle_position == -1  -> set_needle_position = 0, set_signal = 0
    /* when train is entering the tunnel, Killick wants to set telegraph
```

```
      needle’s position to OCCUPIED and no showing flag */
    train_position(i) >= e    -> set_needle_position = 1, showing_flag == FALSE
    /* when the alarm is ringing, Killick wants to set it off and show a flag to indicate situation */
    alarm_ringing == TRUE  -> set_alarm = OFF, showing_flag = TRUE
    /* when Killick wants to showing an indicating flag and a train is
      in the tunnel, he shows the red flag. */
    showing_flag == TRUE && telegraph_needle_position == 1 -> showingFlagColour="RED"
   /* when Killick wants to showing an indicating flag and no train is
      in the tunnel, he shows the white flag. */
    showing_flag = TRUE && telegraph_needle_position == 0  -> showingFlagColour="WHITE"
}
agent Brown {
oracle
    train_position(i) /* will get this agency when the train i is leaving the tunnel. */
    telegraph_needle_position
  handle
    set_needle_position /* set telegraph_needle_position to CLEAR(0) */
  protocol
    trans_position(i) + train_length(i) >=f    -> set_needle_position = -1
}
agent driver(i) {
  state
    pedalAccelerator /* TRUE or FALSE */
    pedalBrake       /* TRUE or FALSE */
    set_train_speed  /* UP, KEEP, DOWN and STOP */
    incrAccPos   /* pedal accelerator further(1), still(0), or less(-1) */
    set_gear
  oracle
    signal_sign  /* the driver gets this agency when he can see the signal */
  showing_flag /* the driver gets this agency when he can see the flag showing by Killick */
handle
  treadlePressed /* to see if the train has press the treadle  */
  signal_treadle  /* set signal_sign to CAUTION(1) or FAIL(-1) when the train passes it */
  brakePos      /* 0 ~ 1 */
  accPos        /* 0 ~ 1 */
privilege
  train_position > b  -> train_press_treadle = @
  train_position <= b  -> train_press_treadle = TRUE
  train_position + train_length <= b  -> train_press_treadle = FALSE
  /* when the train i passes over b point and havn't pressed the treadle,
    press the treadle and make it set the signal or alarm */
  train_press_treadle == TRUE && treadlePressed == FALSE
    -> treadlePressed = TRUE, signal_treadle = ON
  train_press_treadle == FALSE && treadlePressed == TRUE  -> treadlePressed = FALSE
  /* When the driver see ALL_CLEAR sign indicated by signal or a flag,
    he speeds up or keeps the train speed */
  signal_sign == 0 || showing_flag == "WHITE"
    -> set_train_speed = UP or set_train_speed = KEEP
  /* When the driver see CAUTION sign indicated by signal, he slowes
    down the train's speed by reducing his accelerator*/
  signal_sign = 1   -> set_train_speed = DOWN
  /* When the driver see OCCUPIED sign indicated by the red flag, he
    stop the train behind the point d by using his braker*/
  showing_flag = RED -> set_train_speed = STOP
  set_train_speed = UP  -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = 1
  set_train_speed = KEEP  -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = 0
  set_train_speed = DOWN -> pedalAccelerator = TRUE, pedalBrake = FALSE, incrAccPos = -1
  set_train_speed = STOP    -> pedalBrake = TRUE, pedalAccelerator = FALSE
```

```
}
agent signal {
  state
    signalSign        /* CAUTION(1) and ALL_CLEAR(0) */
    currentSignalSign  /* current signal sign        */
    signal_treadle    /* cause the signal to be set to CAUTION(1) */
    treadlePressed
  handle
    signalSign
    set_alarm
  protocol
    /* When the treadle is set to ON by a train and the signal doesn't fail,
      reset the treadle to OFF and set signal to CAUTION. */
    signal_treadle == ON && signalFailure = FALSE  -> signal_treadle = OFF, set_signal = 1
    /* When the treadle is set to ON by a train and the signal does fail,
      set the alarm to ringing and reset the treadle to OFF. */
    signal_treadle == ON && signalFailure = TRUE
     -> set_alarm = ON, signal_treadle = OFF, set_signal = -1
    set_signal == 1   -> signalSign = 1, currentSignalSign = 1
    set_signal == 0   -> signalSign = 0, currentSignalSign = 0
    set_signal == -1  -> signalSign = currentSignalSign
    /* When the treadle is set to ON by a train, it wants to set signal
      CAUTION by a given way (here I assumed it is random) */
    signal_treadle == ON  -> signalState = rand(100), signalFailure = @
    signalState <= signalReliability * 100  -> signalFailure = FALSE
    signalState > signalReliability * 100    -> signalFailure == TRUE
}
agent telegraph {
  state
```

```
    telegraph_needle_position /* OCCUPIED(-1) NEUTRAL(0) CLEAR(1) */
  protocol
    set_needle_position = -1 -> telegraph_needle_position = -1
    set_needle_position = 0   -> telegraph_needle_position = 0
    set_needle_position = 1   -> telegraph_needle_position = 1
}
agent alarm {
  state
    alarm_ringing
  protocol
    set_alarm = ON -> alarm_ringing = TRUE
    set_alarm = OFF -> alarm_ringing = FALSE
}
agent train (i) {
  state
    train_position(i)
    train_speed(i)
    train_length(i)
    gearFw
  derivate
    movedDistance = train_speed(i) * timePeriod + 0.5 * Acc * timePeriod ^2
    train_position(i) = |train_position(i)| + movedDistance * movedDirection
  protocol
    gearFw = TRUE   -> movedDirection = 1
    gearFw = FALSE -> movedDirection = -1
}
```

The following shows various positions of a train passing through the Clayton Tunnel (from the right end to the left end).

← London

```
X<-----X<--------X<-----X<---------X<-----------------X<------X<-
g      f         e      d          c                  b       a
```

point a: start point - it is assumed that the driver can see the signal from this point.

point b: the position of the signal

point c: the position where Killick and the drivers can see each other, this is changeable

point d: the position of Killick

point e: the entrance of the tunnel

point f: the exit of the tunnel where Brown sees the train emerging

point g: the end point where the train disappears from Brown's view.

Appendix 6-B: An Example of a Generic Observable (GO) – *train*

```
%donald
><trainDriver
viewport  ~site
openshape  TRAIN
within TRAIN {
     int trainLength, trainHigh
     trainHigh = ~_SITE_size! div 10
     point train_pos
     int train_pos_x, train_pos_y
     train_pos_x = (~_SITE_size! div 50) + (~_SITE_railLen!)
     train_pos_y = ~_SITE_rail_pos_y!
     train_pos = {train_pos_x, train_pos_y}
     rectangle trainBody
     trainBody = rectangle(train_pos, train_pos + {trainLength, trainHigh})
     label trainLabel
     char  trainNo
     trainLabel = label(trainNo, train_pos + {trainLength div 2, trainHigh div 2})
     ? `"A_" // str(eval(~trainDriver)) // "_TRAIN_trainBody"` = "fill=solid,color=purple";
}
%eden
_TRAIN_trainLength=100 * ~len_ratio;
_TRAIN_trainNo = str(~train_id);
trainStartPos = (float(~_SITE_size) / float(50)) + (~_SITE_railLen);
Killick_see_train = FALSE;
Brown_see_train = FALSE;
driver_see_signal = FALSE;
driver_see_flag = FALSE;
defineCrash = FALSE;
driverSeeSignalPos is eval(~driver_see_signal_pos) * ~len_ratio;
KillickSeeTrainPos is eval(~Killick_see_train_pos) * ~len_ratio;
driverSeeFlagPos is eval(~driver_see_flag_pos) * ~len_ratio;

proc changeTrainAgency : _TRAIN_train_pos_x {
    auto  trainPos, temp, temp1, temp2, i;
    trainPos="_"//str(eval(~trainDriver))//"_TRAIN_train_pos_x";
    if (_TRAIN_train_pos_x <= ~_SITE_Killick_pos + KillickSeeTrainPos &&
         _TRAIN_train_pos_x + _TRAIN_trainLength >= ~_SITE_tunnel_r_pos &&
    !Killick_see_train) {
    addAgency("Killick", "oracle", trainPos);
    Killick_see_train = TRUE;
}
if ((_TRAIN_train_pos_x >= ~_SITE_Killick_pos + KillickSeeTrainPos ||
     _TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_tunnel_r_pos) &&
    Killick_see_train) {
    removeAgency("Killick", "oracle", trainPos);
   sendClient("Killick", trainPos // " = 99999;\n");
    Killick_see_train = FALSE;
}
if (_TRAIN_train_pos_x <= ~_SITE_tunnel_l_pos && !Brown_see_train) {
   addAgency("Brown", "oracle", trainPos);
   Brown_see_train = TRUE;
   sendClient("Brown","autoClearButton();\n");
}
if (_TRAIN_train_pos_x + _TRAIN_trainLength < 0) {
   removeAgency("Brown", "oracle", trainPos);
   Brown_see_train = FALSE;
}
if (_TRAIN_train_pos_x - ~_SITE_signal_pos <= driverSeeSignalPos &&
    _TRAIN_train_pos_x >= ~_SITE_signal_pos && !driver_see_signal) {
   addAgency(eval(~trainDriver), "oracle", "signalSign");
   addAgency(eval(~trainDriver), "handle", "signalSign");
   addAgency(eval(~trainDriver), "handle", "_ALARM_ringing");
   sendClient(eval(~trainDriver), "signalSign = "//str(~signalSign)//";\n");
   driver_see_signal = TRUE;
}
if (_TRAIN_train_pos_x + _TRAIN_trainLength <= ~_SITE_signal_pos &&
    driver_see_signal) {
   removeAgency(eval(~trainDriver), "oracle", "signalSign");
   removeAgency(eval(~trainDriver), "handle", "signalSign");
   removeAgency(eval(~trainDriver), "handle", "_ALARM_ringing");
   driver_see_signal = FALSE;
}
if (_TRAIN_train_pos_x - ~_SITE_Killick_pos <= driverSeeFlagPos &&
    _TRAIN_train_pos_x >= ~_SITE_tunnel_r_pos && !driver_see_flag) {
   addAgency(eval(~trainDriver), "oracle", "_FLAG_showing");
   addAgency(eval(~trainDriver), "oracle", "showingFlagColour");
```

```
        addAgency(eval(~trainDriver), "oracle", "_FLAG_flagpole_pos_x");
        addAgency(eval(~trainDriver), "oracle", "_FLAG_goLeft");
        sendClient(eval(~trainDriver), "_FLAG_showing = "//str(~_FLAG_showing)//";\n");
        sendClient(eval(~trainDriver), "showingFlagColour = \""//str(~showingFlagColour)//"\";\n");
        sendClient(eval(~trainDriver), "_FLAG_flagpole_pos_x =
"//str(~_FLAG_flappole_pos_x)//";\n");
        sendClient(eval(~trainDriver), "_FLAG_goLeft = "//str(~_FLAG_goLeft)//";\n");
        driver_see_flag = TRUE;
      }
      if (_TRAIN_train_pos_x <= ~_SITE_tunnel_r_pos && driver_see_flag) {
        removeAgency(eval(~trainDriver), "oracle", "_FLAG_showing");
        removeAgency(eval(~trainDriver), "oracle", "showingFlagColour");
        removeAgency(eval(~trainDriver), "oracle", "_FLAG_flagpole_pos_x");
        removeAgency(eval(~trainDriver), "oracle", "_FLAG_goLeft");
        driver_see_flag = FALSE;
      }
      if (!defineCrash && ~trainList# > 1) {
        for (i = 1; i <=~trainList#; i++) {
          if (str(~trainList[i]) != eval(~trainDriver))
            ~defineTrainCrash(str(~trainList[i]), eval(~trainDriver));
        }
        defineCrash = TRUE;
      }
}

>>
message = "train_id = \"" // str(train_id) // "\";\n";
message = message // "trainDriver = \"" // str(trainDriver) // "\";\n";
message = message // "include(\"/dcs/res/sun/dtkeden/railway/train.panel\");\n";
sendClient("Killick", message);
sendClient("Brown", message);
append trainList, trainDriver;
```