7 Evaluations and Prospects for EM Tools

In this chapter, we shall explore different techniques and tools that can be used to support the activities of EM. Our objectives are to evaluate possible implementations and to discuss the prospects for future development of EM tools. The chapter starts with a discussion of essential characteristics of an ideal EM tool. This is to set the context for evaluations described in later sections. Section 7.2 describes three existing technologies, Java, Excel and Forms/3, as possible EM tool implementations. Section 7.3 evaluates the principal EM tool: TkEden. Section 7.4 describes a new graphical EM tool, WING, which aims to solve some of the issues of TkEden described in section 7.3. Section 7.5 introduces yet another new EM tool, EME, which aims to explore issues that have not been addressed by WING. In section 7.5, we will highlight some prospects based on research described in previous sections.

7.1 The ideal EM tool

One crucial prerequisite for the successful application of EM is the availability of specialised computer-based tools to build artefacts as EM models. From the discussions in the previous chapters, we know that EM can address issues relating to the whole spectrum from system development to use. Chapter 5 is concerned with the activities that are involved prior to or in the preliminary stages of system development. Chapter 3 overviews the application of EM to system development. Chapter 4 gives an account of the combination of system design and use that is characteristic of EM. Chapter 6 considers the possible role for EM in the use of ubicomp systems. Despite these various different emphases, their underlying EM activities can all be captured by the DMF described in chapter 2. The DMF can be viewed as an idealised framework for EM.

It is useful here to summarise the essential characteristics of the DMF we discussed in chapter 2. They will be used as criteria for evaluating different tool implementations for *conceptual support*. The DMF has the following essential

characteristics:

- **State-based observation** – In the DMF, observation of states is fundamental to experimentation and understanding of dependencies and agencies in the model. This is the basis for EM.

- **Indivisible stimulus-response patterns** – Each definition in the definition sets represents an indivisible stimulus-response pattern. There is no interruption after a stimulus until a response is given.

- **Subjective agency analysis** - Grouping of definitions and actions into agents is subjective to each agent. Therefore, each agent can have their own view of agencies in the model (cf. subject-oriented programming [Har95]).

- **Universal agency** – In the DMF, not only humans but anything can be viewed as an agent. Human and automatic agents are not necessarily distinguished in the framework. In the process of modelling, the modeller can play different roles in views of different agents in the model – a modeller can be an external observer whose interest is only to observe but not modify the model; a modeller can be an actor who sits within the model as one of the agents; a modeller can also be a director (super-agent) who directs the characteristics of other agents.

- **Concurrent agency** – Typically, a model includes lots of different agents. Each agent can act at the same time as others do. Actions can be performed in parallel.

- **Openness of privileges** – Privileges of agents towards definitions and actions of other agents are not circumscribed and can be changed easily and dynamically during the modelling process (cf. OO encapsulation of data).

- **Evolutionary construal** – Construal of observation, dependency and agency is evolutionary. The modelling process typically starts with only observations without agency. Agencies emerge from experiences obtained in the modelling process. Therefore, a model should be constructed interactively.

An ideal EM tool should implement all of these characteristics of the DMF.

However, we shall discuss in this chapter that, in practice there is still no satisfactory implementation of the DMF.

In addition to the concerns of conceptual support, we shall also evaluate interfaces of the tools. We shall investigate interface techniques that make the process of EM easier to perform. Previous tool research in the EM research group has concentrated on the conceptual support where all efforts are on bringing the DMF close to implementation. Concerns about suitable interfaces for an EM tool are relatively unexplored. There are no fixed criteria for interface evaluation. Therefore, evaluations will be driven by our experience of using the tools. We shall conduct more detail evaluation to our principal tool TkEden at interface level in section 7.3 and discuss ideas explored in development of two new tools WING and EME in section 7.4 and 7.5 respectively.

An analogy can be made with the development of the personal computer. Where conceptual support is concerned, an operating system gives a faithful support for the Von Neumann machine framework. Where the interface is concerned, graphical user interfaces make an operating system easy to use. The success of the personal computer is not possible without a well-developed graphical user interface, and the success of the graphical user interface is not possible without a well-developed operating system. Therefore, we believe that both concerns are very important. We should address both concerns in order to enhance practical applications of EM. By doing so, our main aims are:

- To explore possible options to improve applications of EM in practice (i.e. narrowing the gap between theory and practice).
- To enhance user experience for the process of EM and make EM more suitable to novice users
- To obtain insights on possible options of interfaces for improvement of current tools or development of new tools in the future.

Throughout this chapter, we shall implement a simple DMF model (called Business Deal) by using different techniques and tools. We shall use these implementations as examples used by evaluation of different techniques and tools against the essential characteristics of the DMF. The DMF model of Business Deal is shown in Figure 7.1.

```
Seller's definitions:
asking_price is 150

Buyer1's definitions:
budget is 100
interested is Seller:asking_price<=budget

Byer1's latent actions:
(interested = true) -> Seller:Buyer1Offer is true
(interested = false) -> Seller:Buyer1Offer is false

Buyer2's definitions:
budget is 1000
interested is Seller:asking_price*0.2<=budget

Buyer2's latent actions:
(interested = true) -> Seller:Buyer2Offer is true
(interested = false) -> Seller:Buyer2Offer is false
```

Figure 7.1: The Business Deal model in the DMF (note that this is only a conceptual representation)

There are a `Seller` agent and two `Buyers` agents in the model. The `Seller` agent tries to sell his product for a specified asking price. Both `Buyers` have a budget and a criterion to determine if they are interested in making an offer (by actions) for the asking price. In this case, `Buyer1` is interested in making an offer only if the asking price is below or equal to his budget. For `Buyer2`, only if the asking price is below or equal to twenty percent of this budget. Therefore, in the whole model, there is one definition for the `Seller`, and two definitions and two actions for each of the Buyers.

There is no doubt that any of the techniques and tools discussed in this chapter can implement this scenario. However, our main evaluation aim is to investigate the *directness of translation* from the DMF model to particular implementations. By analogy, we can devise OO programs merely by using a procedural programming language (e.g. using C). However, it is far more effective if concepts of OO can be directly supported by the programming language.

7.2 Existing technologies as possible EM tool implementation

In this section, we shall explore three existing technologies as possible EM tool implementations. First, we shall experiment with model-building using a typical OO language: Java. Second, we shall experiment with a spreadsheet application: Excel. Finally, we shall experiment with a visual first-order functional language, Forms/3.

7.2.1 Model-building using Java

Java is a language specially designed for OO programming. However, in this section we shall implement the Business Deal model using Java (JDK version 1.3.1). We represent agents as *objects*, observables as *attributes* and actions as *methods*. Since there is no direct representation for definitions, we have made use of `Observable` and `Observer` classes (from the standard Java library in `java.util`) to simulate the maintenance of dependencies. The mechanism is as follows. Any class that inherits from the `Observable` class maintains a list of objects that implements the `Observer` interface. An `Observable` object can notify `Observer` objects in the list whenever changes have occurred by calling its method `notifyObservers`. This method calls `update` methods of each `Observer` objects in the list as a notification for the change.

As shown in Listing 7.1, `Buyer1` (lines 1-14) and Buyer2 (lines 15-28) agents are represented by classes implementing the `Observer` interface. The major differences between `Buyer1` and `Buyer2` are their criteria of making an offer (lines 9 and 23).

```
1.   class Buyer1 implements Observer{

2.   private double budget;
3.   private boolean interested;

4.   Buyer1(){
5.     budget=100;
6.   }

7.   public void update(Observable o,
     Object arg){
8.     Seller s=(Seller)o;
9.     interested=s.getPrice()<=budget;
10.    if(interested){
11.    s.offer("Buyer1");
12.    }
13.  }
14.  }
```

```
15.  class Buyer2 implements Observer{

16.  private double budget;
17.  private boolean interested;

18.  Buyer2(){
19.    budget=1000;
20.  }

21.  public void update(Observable o,
     Object arg){
22.    Seller s=(Seller)o;
23.  interested=s.getPrice()<=budget*.2;
24.    if(interested){
25.    s.offer("Buyer2");
26.    }
27.  }
28.  }
```

Listing 7.1: Two classes representing Buyer1 (left) and Buyer2 (right) agents.

As shown in Listing 7.2, the `Seller` (lines 29-42) agent is represented by a class inheriting from the `Observable` class. The `Deal` class is a dummy class that contains a program entry point (i.e. the `main`).

```
29.  class Seller extends Observable{

30.  private double asking_price;

31.  public void setPrice(double price){
32.    asking_price=price;
33.    setChanged();
34.    notifyObservers();
35.  }

36.  public double getPrice(){
37.    return asking_price;
38.  }

39.  public void offer(String sellerName){
40.    System.out.println(sellerName +"
     made offer.");
41.  }
42.  }
```

```
43.  class Deal{

44.  public static void main(String
     arg[]){
45.     Seller seller=new Seller();
46.     Buyer1 buyer1=new Buyer1();
47.     Buyer2 buyer2=new Buyer2();

48.     seller.addObserver(buyer1);
49.     seller.addObserver(buyer2);

50.     System.out.println("Price set to:
     10");
51.     seller.setPrice(10);
52.     System.out.println("Price set to:
     150");
53.     seller.setPrice(150);
54.     System.out.println("Price set to:
     999");
55.     seller.setPrice(999);
56.  }
57.  }
```

Listing 7.2: Seller agent (left) and the testing class (right).

A typical scenario of running this program is as follows. The main loop instantiates Seller, `Buyer1` and `Buyer2` (lines 45-47). Then, dependencies between the `Seller` and `Buyers` are made by adding `Buyers` to the list of `Observers` in the `Seller` (lines 48-49). After that, we can then test the responses of `Buyers` by setting different prices in the Seller. For example, the program calls `seller.setPrice(150);` (line 53). This sets the price in the Seller to 150 and notifies `Buyer1` and `Buyer2` (lines 31-35). The `update` methods in `Buyer1` (line 7-14) and `Buyer2` (line 21-28) are called. By their criteria, `Buyer1` does not make offer but `Buyer2` does. The output of running this programming is shown in Figure 7.2 below.

```
C:\>java Deal

Price set to: 10
Buyer1 made offer.
Buyer2 made offer.
Price set to: 150.
Buyer2 made offer.
Price set to: 999.
```

Figure 7.2: Output of running the Business Deal model in Java

Building an ‘EM model’ by using Java is difficult. The first very noticeable problem is we cannot build the model interactively – every time we change or add something to the model, we need to make a compilation and restart the test from the beginning. We cannot add or modify any definition or action during the execution of the Java model. This is not in keeping with the evolutionary characteristics of the

DMF where by modification of the model should be conducted interactively as one of the agents.

User interaction with the model has to be circumscribed before executing the model. So the user cannot be an agent within the model (cf. universal agency in the DMF). In fact, interaction between other agents in the model also has to be circumscribed.

We could represent observables by attributes which contain states of the agents. However, the dependencies between observables from different agents have to be maintained by message passing (cf. line 34 and line 9). In other words, indivisible stimulus-response patterns cannot be represented faithfully in a Java model. Besides, the focus on states is easily inhibited by paying too much attention to message passing between classes (cf. state-based observation in the DMF).

The concurrency of actions of three agents in the Java model is replaced by sequential message passing (cf. actions in lines 11 and 25). We can implement concurrency explicitly in Java but this involves setting up the mechanism manually as part of the program. What we really need is the style of concurrency that exists in the DMF (cf. concurrent agency in the DMF).

In OO philosophy, it is better to declare attributes of a class as `private` (i.e. data encapsulation). However, in the DMF every observable of an agent is accessible for other agents. This can be easily done by declaring all attributes of a class as `public`. It violates OO philosophy but facilitates the DMF philosophy (cf. openness of privileges in the DMF).

All the above observations lead us to conclude that it is difficult to use Java to build an EM model. The representation of agents as objects, observables as attributes and actions as methods seem to be direct but in fact it is not. The translation from the DMF model to the Java model is neither direct nor complete.

7.2.2 Model-building using Excel

In this section, we describe building the Business Deal model by using the spreadsheet application Excel (version 2002). In Excel, spreadsheets are called ‘worksheets’. Worksheets are grouped together as a ‘workbook’. We can represent observables as cells, dependencies as formulae, actions as macros (in Visual Basic),

agents as aggregations of worksheets and macros, and the entire model as a workbook.

Figure 7.2 shows screen captures of the model. The Seller agent represented by a worksheet is at the bottom-right screen capture. The current asking price is in the cell B1. Two Buyers are two worksheets (in formulae view) at the top of the figure with their budget and criteria of making an offer. We can see in the figure how each worksheet can refer to cells in another worksheet by using the symbol '!'.

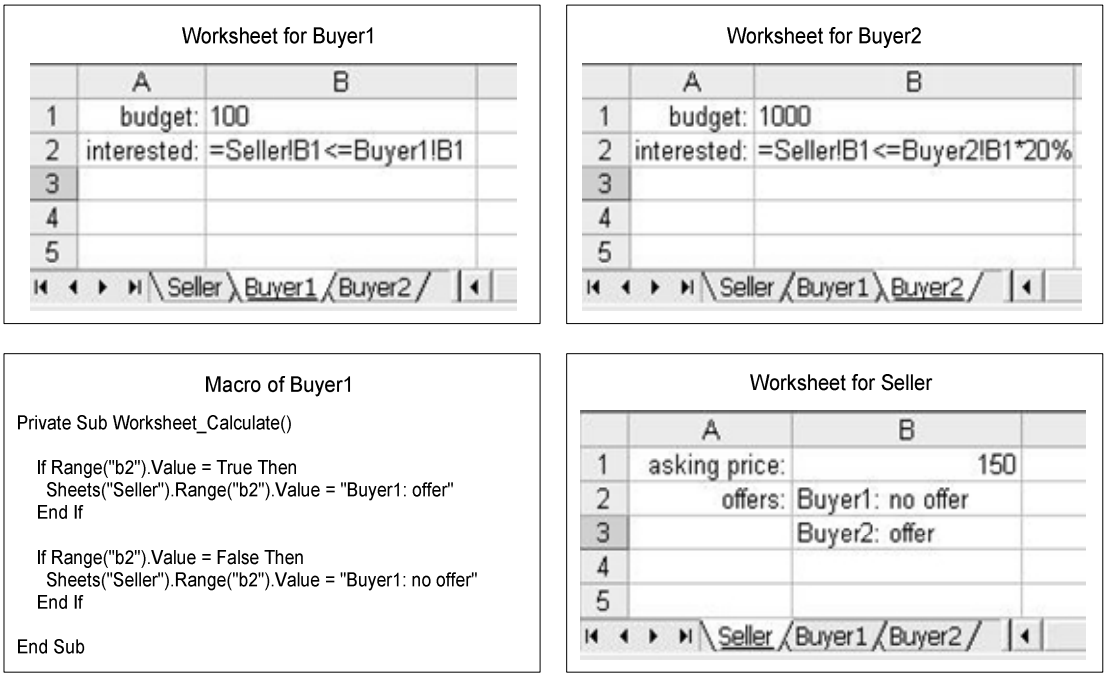

Figure 7.2: The Business Deal model in Excel

At the bottom-left screen capture of the figure shows an 'event macro' called `Worksheet_Calculate` attached to Buyer1 worksheet. Every worksheet in Excel has this macro by default. Any code specified in this macro is automatically executed whenever any cell value in the worksheet is updated. As shown in the figure, it contains actions of Buyer1 – that changes the value of Seller's B2 cell to make an offer. A similar macro is also attached to Buyer2 which is not shown in the figure.

The translation from the DMF model to a spreadsheet model is more direct than to the Java model. This is mainly for two reasons: dependencies specified by formulae are automatically maintained (cf. indivisible stimulus-response pattern in the DMF) and we can incrementally build the model on-the-fly (cf. evolutionary in the DMF).

By using the grid layout, a spreadsheet visualises all states corresponding to cells (cf. state-based observation in the DMF). By changing from one worksheet to another, the user can play different roles as different agents in the model (cf. universal agency in the DMF).

Excel provides no support for specifying access privileges for the cells. Every worksheet is free to refer to cells in other worksheets (cf. openness of privileges in the DMF). We have also demonstrated by experiments that macros (actions) are executed sequentially in Excel. Therefore, agents cannot act concurrently (cf. concurrent agency in the DMF).

To conclude, we can use Excel as an EM tool. The translation from the DMF model to the spreadsheet model is easy. There are direct correspondences in mapping agents as worksheets, cells as observables, definitions as formulae and actions as macros. However, we have also notice two discrepancies. First, the agents cannot behave concurrently. Second, observables are normally referenced by grid co-ordinates rather than by user-given names (we can give an alias to cells but this involves extra effort). In order to use Excel as an EM tool, we also need to extend its available cell data types; it would be especial helpful to introduce graphic data types.

7.2.3 Model-building using Forms/3

Forms/3 is a first-order functional visual programming language based on spreadsheet styles of cells and formulae [Bur01]. However, there are two main differences between Forms/3 and a conventional spreadsheet. First, in Forms/3, cells are represented as rectangles which are positioned manually without a conventional grid layout. The value of a cell is visualised within the cell. The name and a *formula tag* of the cell are located at the bottom of the cell. Clicking the formula tag brings up a dialogue in which a formula can be entered (see left-hand-side of Figure 7.3). Second, in conventional spreadsheet, we can change the value of a cell by using procedural macros. In Forms/3 the value of a cell is solely defined by its formula (this is termed the ‘value rule’). Hence, it does not support procedural macros. This is consistent with one of the main aims of Forms/3: to stay within the functional paradigm of programming.

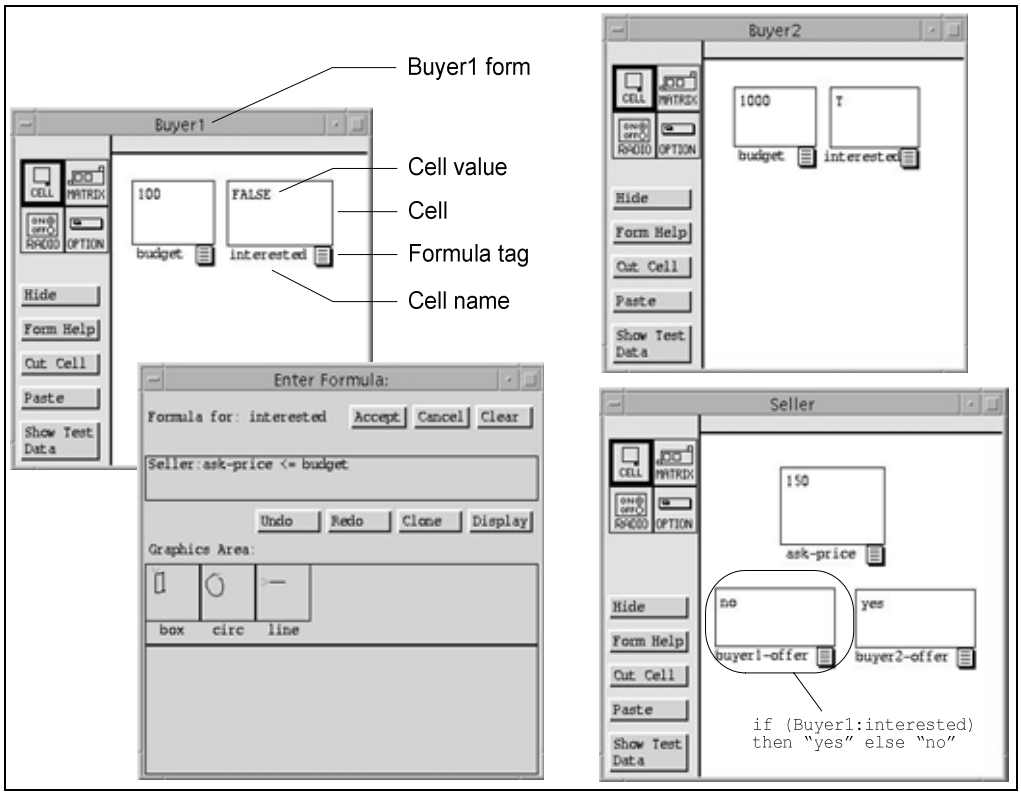

Figure 7.3: The Business Deal model in Forms/3

Figure 7.3 shows a Forms/3 implementation of the Business Deal model. We can represent agents as *forms* (similar to worksheets in Excel), observables as cells and dependencies as formulae. Because of the value rule, there is no way we can specify actions directly. Therefore, we cannot specify the 'offer-making' actions in two Buyers. The workaround was to translate actions into definitions. For example, instead of specifying the actions in Buyer1, we enter an observable in the Seller with the formula: `if (Buyer1:interested) then "yes" else "no"` (bottom-right region of Figure 7.3). However, by doing this we have changed the semantics of the model – in the DMF model `offer` is *initiated* by the Buyers but in the Forms/3 model `offer` is *perceived* by the Seller.

By using Forms/3, we can perform state-based observation, specify indivisible stimulus-response patterns and construct models evolutionarily (just like Excel). Cells can also be grouped to reflect agencies as forms. However, the inability to specify actions makes translation from the DMF model to Forms/3 model difficult and sometimes impossible without changing the semantics of the model. This limitation of using Forms/3 as an EM tool highlights the difficulty in building an EM model in using pure functional constructs.

7.3 TkEden: the principal EM tool

Our principal tool TkEden and its variants (e.g. ttyEden, Eden, DTkEden) have been used for EM for more than ten years. Over this period, the research interests of EM itself have broadened from originally being solely concerned with interactive graphics [Bey88b] to definitive programming [Yun93], and from definitive programming to system development. To justify our claims that EM is not only theoretical but also practical, the expansion in conceptual scope of EM needs to be accompanied and supported by developing practical tools. TkEden serves very well as a research prototype for trying and exploring different research ideas – several hundreds of models have been built and more than seven different definitive notations have been integrated into the tool. However, in this section, we shall take a critical view of TkEden in use, and identify its limitations as a tool for applying EM in practice.

In a TkEden script, observables are represented by definitive variables. Dependencies are explicitly specified by formulae. Actions are represented by procedures. Agents can be represented by the grouping of definitions and actions in the script, suitability annotated with comments. This is illustrated in Listing 7.3, which shows the script of the Business Deal model. The primary input mechanism is to write a script of definitions in a text input window and then introduce the script into the model by clicking the 'Accept' button of the 'input window' (cf. Figure 2.7 in chapter 2).

```
1.   /*** seller ***/
2.   seller_asking_price is 150;

3.   /*** buyer1 ***/
4.   buyer1_budget is 100;
5.   buyer1_interested is  seller_asking_price <= buyer1_budget;

6.   proc buyer1_action1: buyer1_interested{
7.       if(buyer1_interested) seller_buyer1Offer is 1;
8.   }
9.   proc buyer1_action2: buyer1_interested{
10.      if(!buyer1_interested) seller_buyer1Offer is 0;
11.  }

12.  /*** buyer2 ***/
13.  buyer2_budget is 1000;
14.  buyer2_interested is seller_asking_price <= buyer2_budget*0.2;

15.  proc buyer2_action1: buyer2_interested{
16.      if(buyer2_interested) seller_buyer2Offer is 1;
17.  }
18.  proc buyer2_action2: buyer2_interested{
19.      if(!buyer2_interested) seller_buyer2Offer is 0;
20.  }

21.  proc monitor:seller_buyer1Offer,seller_buyer2Offer{
22.      writeln("buyer1:",seller_buyer1Offer," buyer2:",seller_buyer2Offer);
23.  }
```

Listing 7.3: TkEden script of the Business Deal model

We shall first evaluate the degree of conceptual support for EM provided by TkEden by investigating the directness of the translation from a DMF model to a TkEden model. After that, we shall evaluate its interface to investigate the ease-of-use aspects.

7.3.1 Evaluation of conceptual support

Conceptually, the TkEden interpreter provides a high degree of interactiveness, so that the modeller can gradually build the model on-the-fly (cf. evolutionary construal in the DMF). Any changes are immediately reflected in the model. The modeller can use a trial-and-error approach to experiment with the model. The states of observables can be queried at any time during the modelling process, primarily by using a "`writeln();`" statement (cf. state-based observation in the DMF). The dependencies specified by using formulae are automatically maintained (cf. indivisible stimulus-response patterns in the DMF). Every observable is in a global name space, and therefore can be accessed freely from anywhere in the model (cf. openness of privileges in the DMF).

The only major discrepancy between the DMF model and the TkEden model is in the representation of agency. TkEden has no direct mechanism to represent agents. As shown in Listing 7.3, we can use comments to annotate the agents (e.g. "`/*** buyer1 ***/`" in line 3) and use naming conventions to specify the ownership of observables (e.g. `buyer1_budget` in line 4). However, these ways of representing agency can only be interpreted by human agents. We need some built-in agency representation mechanisms that are interpretable both by human and automatic agents.

In addition, concurrent actions can only be executed one by one sequentially (cf. concurrent agency in the DMF). Modelling a real-time system which has agents acting concurrently becomes very difficult. This is illustrated in the Dishwasher model described in chapter 3. In the Dishwasher model, a clock agent is introduced to simulate concurrency. However, all actions are still executed sequentially. Efforts have been made in the distributed version of TkEden (DTkEden) to allow concurrency through *actions* [Sun99a]. However, there is still no mechanism to allow *dependencies* to be maintained concurrently.

Another issue is that TkEden is a hybrid interpreter that allows both procedural

variables and definitive variables to be used. The modeller has to distinguish a procedural assignment from a definition. For example, "`a = b+c;`" and "`a is b+c;`" are very different. In the former, `a` is a procedural variable. Only the result of the calculation from `b+c` is assigned to `a`. This is a one-off calculation. Value changes to `b` or `c` do not cause the value of `a` to be changed. In the latter, `a` is a definitive variable. The value of `a` will be recalculated every time the value of `b` or `c` changes. In the DMF, we can have only definitive variables, and assignment such as "`a = b + c;`" is replaced by "`a is eval(b+c);`" where `eval()` returns the 'current value' of "`b+c`". The use of procedural variables in TkEden is sometimes confusing, and it violates the characteristics of the DMF.

7.3.2 Evaluation of the interface

In this subsection, we evaluate TkEden for issues of ease of use. The result of the evaluation can give useful insights into how to build better interfaces for EM tools. With reference to the analogy with the development of the personal computer mentioned earlier, TkEden's interface is like a command prompt interface to an operating system – it provides access to all functionalities of the system but it is *not* the best access method for end-users when compared to graphical interfaces. In addition, some characteristics of the DMF can be more easily realised at the interface. For example, the process of subjective agency analysis can be facilitated by allowing the modeller to organise definitions and actions visually. This also supports the conception of the model as an artefact.

TkEden has two features that are related to ease of use. Firstly, it promotes high-level, task-specific programming style that allows the modeller to construct a model interactively by trial-and-error. This is a common feature in most end-user programming tools (cf. [Nar93]). Secondly, the Eden interpreter that is kernel of TkEden provides dynamic typing, so that definitive variables need not be declared before use. Apart from these two features, TkEden has many issues that need to be addressed in relation to ease of use. We shall discuss them one by one as follows:

- **The definitive script is not the interface –** The use of definitive script in TkEden is limited to model representation. It is *not* the interactive model itself – we cannot change the state of the model by editing the script directly. A definitive script has to be fed into TkEden by using the input window. The actual model resides in the computer memory and can only be accessed through the

'input window'.

- **Lack of default visualisation –** TkEden enables the modeller to build customised visualisations by using Donald and Scout. However, there are typically many elements of a definitive script that are only accessible to the modeller in textual form and typically only as a result of action on the modeller's part. For instance, such elements typically include the values and relationships associated with scalar variables that underlie a geometric model. By default the TkEden interpreter only presents the current values and definitions of such variables on request in response to queries of the form "`writeln(x);`" and "`?x;`". This is unlike a spreadsheet, where all the current values are displayed all the time, and dependencies between cells can be displayed graphically on demand. EM advocates the idea of modelling by building of artefacts. Supplying a default visualisation for all the elements of a model could give the modeller a more concrete feeling of building a model as an artefact.

- **Lack of mechanism to organise definitions** – There is no feature in TkEden to organise definitions and actions in accordance with user preferences. Definitions and actions can only be viewed in the predetermined ways that the interpreter allows (e.g. Eden, Scout and Donald variables can be listed in alphabetical order).

- **Difficulties in renaming observables –** TkEden provides no feature for renaming an existing observable. This involves renaming all the occurrences of the observable in formulae of all definitions in the model. This makes observables 'sticky' because if the modeller wants to rename an observable, she has to redefine all other observables that depend on it.

- **Invisible progress of actions** – In general, the modeller does not need to know the progress of actions. However, when it comes to understanding the behaviour of a model, it is useful if the modeller can inspect (for example) the order of action execution. This also makes debugging easier.

- **Difficulty in annotating definitions** – Comments can be added to definitive scripts. However, when a script is interpreted, all the comments are discarded by the interpreter. Comments help to understand the model so it would be useful to be able to attach comments to definitions even after they have been interpreted. One of the difficulties encountered in annotating scripts is that a comment may

apply to a group of definitions, but there are in general many different useful ways of grouping definitions (cf. subjective agency analysis).

- **Notation syntax inconsistency** – In TkEden, different definitive notations have different syntax. For example, every statement in Eden has to end with a semicolon but in Donald every statement has to be terminated with a carriage-return. Another example is that the user does *not* have to declare variables in Eden but does have to declare them in Donald and Scout. It is difficult and confusing for the novice to learn several different syntactic conventions.

- **Complex syntax for actions** – The syntax for specifying actions is sometimes very complex. An example will be given in discussing the EME tool in subsection 7.5.2.

The developments of WING and EME described in the next two sections aim to address some of these issues.

7.4 WING: a graphical EM tool

WING (a WINdowing and Graphics tool) was originally conceived as the implementation of a definitive notation for windowing and graphical objects. However, during its development, the focus of attention shifted to its capabilities as a full EM tool. WING provides 15 data types in four categories, namely *basic*, *windowing*, *graphics* and *vector* data types. The modeller can also define new data types and operators. In this section, we shall only discuss features of WING that address some of the issues for TkEden identified in the last section. For more details, the reader is directed to [Won98]. Appendix F contains a technical overview of WING.

7.4.1 Organising definitions

Figure 7.4 shows the main user interface of WING loaded with a model of a room. The interface resembles a file explorer – definitions are organised within *containers* in much the same way that files are organised within directories. Therefore, conceptually, agents can be represented by containers (cf. the lack of agency representation and mechanism to organise definitions in TkEden). We call this a *directory-like*

organisation of agents. Clicking a particular container at the left-hand-side of the interface makes all definitions in the container appear at the right-hand-side of the interface. Definitions are displayed as rows of spreadsheet-like cells and can be edited directly by double clicking the cells (cf. definitive script is not interactive in TkEden). The values of observables can be viewed by clicking the corresponding cells (cf. the lack of default visualisation in TkEden). Comments can be added to each definition and these are recorded is displayed by WING (cf. difficulty in annotating definitions in TkEden). This is illustrated in figure 7.4 below.

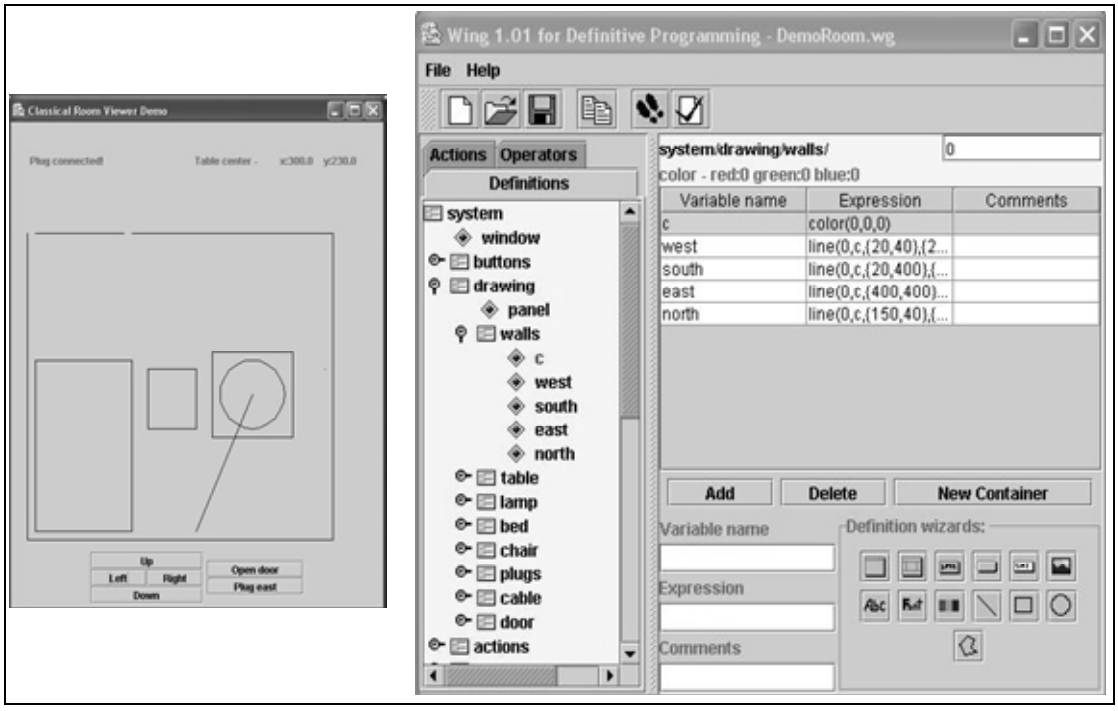

Figure 7.4: WING (right) loaded with the Room model

In addition, WING has a method of alleviating the syntactic obstacles to entering definitions. At the bottom-left area of the interface shown in Figure 7.4, there is a set of buttons for helping the user to enter definitions. Clicking a button brings up a *definition wizard*. Figure 7.5 shows a definition wizard for specifying a line definition. A definition wizard provides a reminder of the syntax for a definition and an explanation of the significance of each parameter. By filling the form, the definition will be automatically generated. This is particularly useful for the novice modeller because she does not have to remember the syntax of all the different types of construct within different definitive notations (cf. notation syntax inconsistency in TkEden).

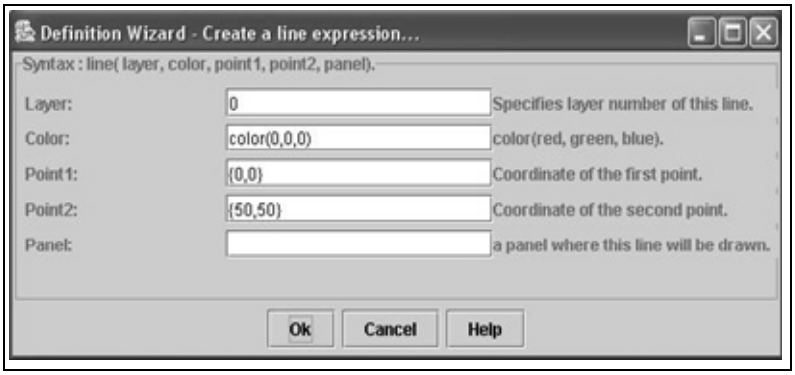

Figure 7.5: A Definition Wizard of specifying a line definition

7.4.2 Specifying actions

Actions are specified by using Java. WING provides a *code template* as the skeleton of an action. The user is only required to fill in actions to manipulate definitions in the model as shown in Figure 7.6. Each action in WING has a priority number which can be changed to control the order of action execution. The actions can be compiled and dynamically linked to the model by clicking the 'Validate' button. Dynamic linking of actions to the model is important because it allows user to specify actions on-the-fly without sacrificing the interactiveness of the model.

The execution of actions triggered by interaction with the model can be monitored by bringing up the action queue window as illustrated in the bottom-left window of Figure 7.6. In the figure, we can see four actions waiting in the action queue. The number at the right of each action name is the priority number. The modeller can step through the execution of actions by clicking the 'step' button at the bottom of the action queue window. This feature addresses the issue of invisible progress of actions in TkEden.

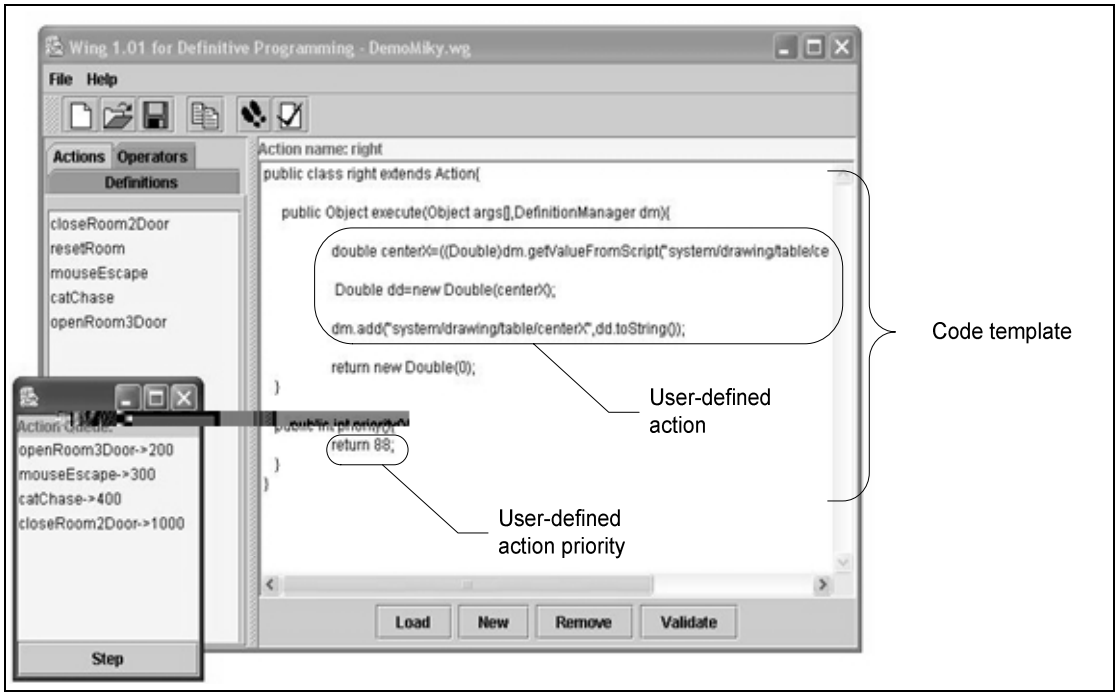

Figure 7.6: Action specification in Java (middle) and stepping for action queue (bottom-left window)

7.4.3 Higher-order definitions

WING also explored the possibility of introducing *higher-order definitions* to an EM model by providing a 'like' operator (cf. the 'like' operator in Excel). A higher-order definition is a definition whose *formula* depends on the formula of another definition. In TkEden, formulae in definitions can only depend on the *values* of other definitions. To illustrate the concept, we can refer to the model of a simple two-layer perceptron (in the context of neural network). In this case, we are interested in the shape of the three neurons. As shown in the left-hand-side of Figure 7.7, originally they are all oval in shape. The task is to change them into a rectangular shape. By using the 'like' operator, we can specify their shapes as follows:

```
centreA is {130,90}
centreB is {130,290}
centreC is {330,190}
neuronA is oval(0, color(0,0,0), centreA, size, panel)
neuronB is like(neuronA,1, centreB, centreA)
neuronC is like(neuronA,1, centreC, centreA)
```

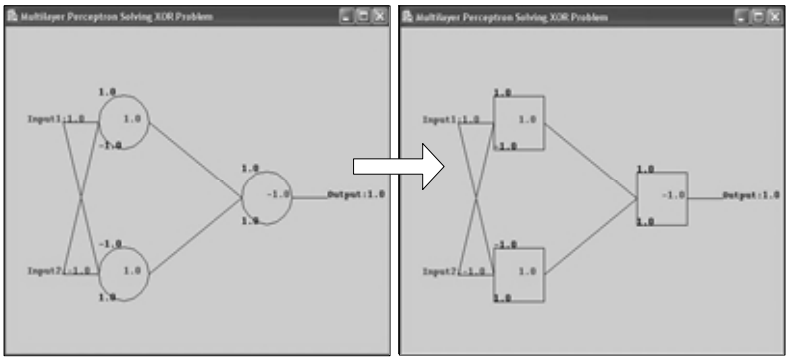

Figure 7.7: Neurons in oval shape (left) are transformed to rectangular shape (right) by higher-order definitions

The centre positions of the neurons are specified by `centreA`, `centreB` and `centreA` respectively. As we can see in the above definitions, the shape of neuron B and C is dependent on the shape of A. For example, in the definition of neuron B: "`like(neuronA,1, centreB, centreA)`" means that the formula of B is same as A *except* that the first occurrence of symbol `centreA` is replaced by `centreB` (because B is in a different position). To change all the shapes to rectangles, we only need to change formula of A from "**`oval`** `(0, color(0,0,0), centreA, size, panel)`" to "**`rect`** `(0, color(0,0,0), centreA, size, panel)`".

WING has illustrated that simple higher-order definition can increase the expressive power of definitions. However, we need to bear in mind that higher-order definitions are more difficult to understand than ordinary definitions. We need to find a balance between expressiveness and comprehensibility in the future development of higher-order definitions in EM tools.

7.4.4 Evaluation

The development of WING has provided possible solutions to most of the issues of TkEden at the interface level. In addition, it has also explored some new ideas such as definition wizards, dynamic compilation of Java code for extension, and higher-order definitions. However, the design and implementation of WING also raises a number of issues:

Firstly, there is no concurrency of agents. We can represent agents as containers. Ideally, actions with the same priority need to be executed in parallel (cf. concurrent agency in the DMF) but in WING they are executed sequentially. A similar issue

arises in TkEden.

Secondly, we cannot simulate a clock. In TkEden, we can simulate concurrency by time-sharing regulated by a clock agent (cf. the use of the `todo` construct Dishwasher model). The implementation of the dependency maintainer in WING does not allow a similar clock agent to be defined. One possible solution is introducing a built-in clock agent in a separate program thread.

Finally, the use of Java language for action specification makes WING more extendable. However, it is difficult for novice modellers to specify functions and actions if they do not have experience in Java programming. We have found that it is more difficult to specify actions in WING than in TkEden. One of the aims for developing EME, to be described in the next section, is to explore the possibility of designing a simpler language for specifying actions.

7.5 EME: a tool with expressive variable referencing

The primary aim of building the Empirical Modelling Environment (EME) was to develop a more expressive language for variable referencing. This is motivated by the need for simpler methods of specifying complex definitive script in TkEden. The variable referencing techniques also lead to alternative techniques for organising definitions. Though EME only supports simple data types, it addresses potential solutions to problems of script construction and management that are relevant to all definitive notations. Appendix G contains a technical overview for the tool.

7.5.1 Entering definitions

Like TkEden, EME accepts textual definitive script from an input window. But unlike TkEden, procedural variables and assignments are not allowed. The typical example of “a is b plus c” can be specified as:

```
a = b + c;
```

The symbol ‘`=`’ here has the same semantics as the symbol ‘`is`’ in TkEden. The semicolon at the end marks the end of a definition. Figure 7.8 shows the interface with some definitions in the model.

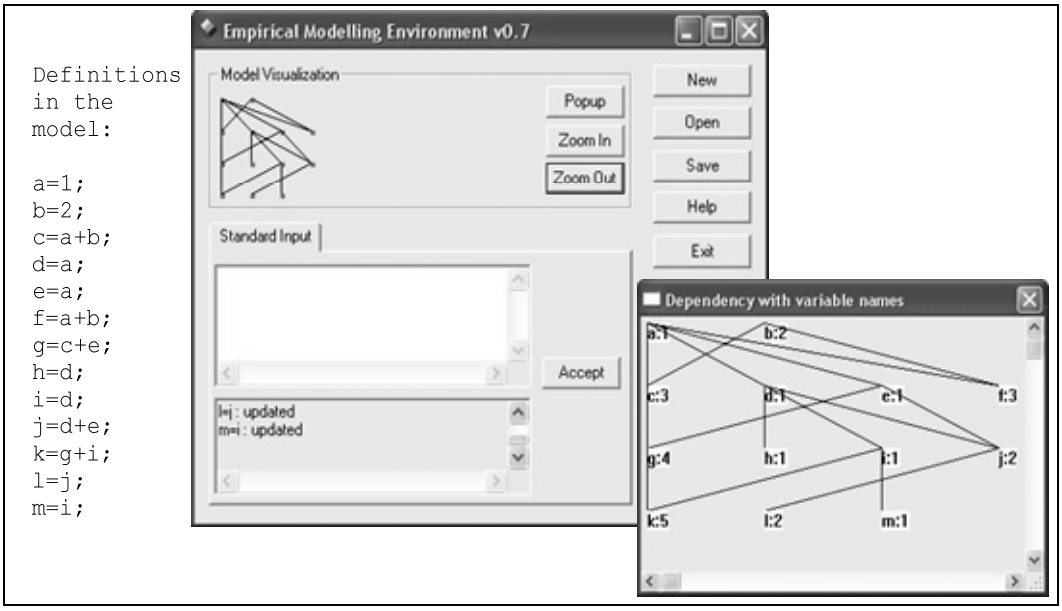

Figure 7.8: EME interface (left) and model visualisation (right)

As shown in the figure, the model is visualised as a hierarchical dependency graph. There are two views of the same graph. The smaller one (on the left in the figure) gives an overall visualisation of the entire model. The bigger one can be zoomed to display a particular part of the graph. The symbols in the graph are ordered in levels so that a variable is only dependent on the variables at levels on top of it. The integer value at the right of each symbol is the current value of that symbol. This feature of EME addresses the issue of lack of default visualisation of the model in TkEden. It gives the modeller a more concrete feeling of building an artefact. This feature has been further enhanced in the development of the DMT to be described in the next chapter.

7.5.2 Variable referencing

One of the unique features of EME is its variable referencing mechanism. This gives the modeller more control of how a variable reference in a definition will be evaluated. The format of a variable is shown in the following example:

```
table\lamp\bulb
```

The backslashes in this variable are context separators. In EME, variables are specified by *variable expressions*. A variable expression may contain alpha-numeric characters, back slashes, and three kinds of operators designated by angle brackets, square brackets and curly brackets.

The angle brackets < > specify that the enclosed expression will be evaluated and translated into alpha-numeric characters and then substituted into the variable expression to form a variable reference. It is a one-off translation. The square brackets [] specify that the enclosed expression will be evaluated and translated into alpha-numeric characters every time the definition is evaluated. Finally, the curly brackets { } specifies that we need the value of the enclosed expression only. To help understand the semantics of those brackets, here is an example. Suppose we already have the following definitions in the model:

```
i = 10;
x\10 = 99;
x\i = 88;
```

The examples in Figure 7.9 show that we can use different brackets to control exactly when a variable expression is evaluated.

User input	Definition stored in the symbol table	Evaluation sequence
y = x\10;	y = x\10	x\10 → 99
y = x<i>;	y = x\10	x\10 → 99
y = x[i];	y = x[i]	x[i] → x\10 → 99
y = x\i;	y = x\i	x\i → 88
y = {x\10};	y = 99	99

Figure 7.9: Examples of using different variable referencing techniques

We can identify three different uses of this technique:

1) Creating a virtual list – We can create a virtual list by using this technique. For example, the following definitions specify a list of three integers:

```
a\2 = 20;
a\3 = 30;
x = a[i];
i = 1;
```

After introducing these definitions, the value of `x` will be changed according to the value of the index `i`.

2) Syntactical organisation of agents – We can specify the ownership of variables by integrating the agent name into the variable name (cf. the concept of 'virtual agent'

in DTkEden [Sun99a]). For example, the following three definitions specify the exam marks for three students:

```
tommy\marks = 100;
annie\marks = 40;
jody\marks = 10;
```

We can refer to each student's marks by the variable expression `<currentStudent>marks` where `currentStudent` contains a name of a student.

3) Simplifying syntax of action specification – This way of referencing definitive variables is neater than the way used in TkEden. For example, here is an action taken from the TkEden timetabling model. Its purpose is to link cell contents to a function `f(x)`.

```
proc linkCells{
  auto i;
  for(i=1;i<=slotsperday*noofdays;i++){

   execute(
   "~slotsTable_" // ~slotsTable_myList[i] // "_myCaption is ~f(" // str(i) // ");");
  }
}
```

The EME implementation of the same action would be:

```
action linkCells{
  i=1;
  repeat(i<=slotsperday*noofdays){

   slotsTable<slotsTable_myList<i>><"myCaption"> = f({i});

  i={i+1};
  }
}
```

If the value of `i` is 1, the definition in the above loop is equal to:

```
  slotsTable\cell1\myCaption = f(1);
```

This is arguably more understandable than the TkEden implementation. The use of keyword `repeat` will be explained in the following subsection.

7.5.3 Procedural macro

EME provides two procedural macros to automate the variable defining process. They are a conditional if macro and a looping macro. The syntax of the conditional if is:

if (logical expression) {lines of definitions} **else** { lines of definitions}

The syntax of the looping macro is:

repeat (logical expression){lines of definitions}

The loop repeats until the logical expression evaluates to be false. To illustrate, we can define five windows by the following definitions. The result is shown in Figure 7.10.

```
i=1;
title="hello!";
repeat (i<=5) {
x1<i> ={10+i*50};
y1<i> ={10+i*20};
x2<i> ={150+i*50};
y2<i> ={400+i*20};
w<i> = window(title,x1<i>,y1<i>,x2<i>,y2<i>);
i = {i + 1};
}
```

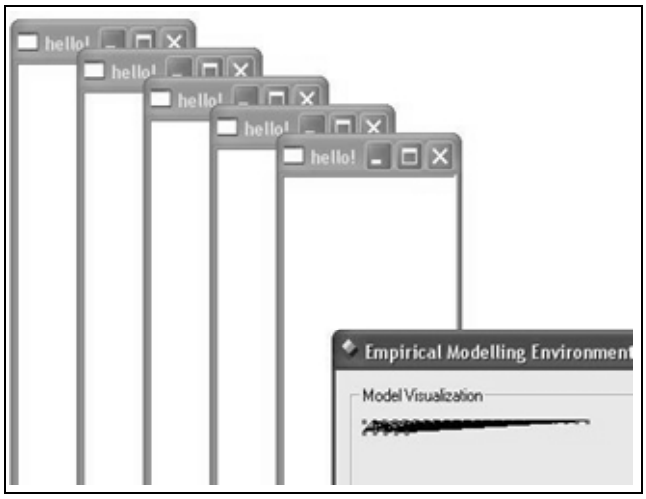

Figure 7.10: Specifying multiple windows

The window at the bottom right of Figure 7.10 illustrates how a simple iteration in EM can generate a very dense set of dependencies.

7.5.4 Complex interface widget

As illustrated in Figure 7.10 and 7.11, EME, like WING, includes features for defining windows and buttons. However, in the development of EME, we have also experimented with the introduction of complex interface widgets. For example, the following two definitions define the a simple spreadsheet grid with 8 by 10 cells depicted in Figure 7.11 (other numbers in the definitions are for specifying the size of the display area).

```
wg = window("grid test",10,10,400,400);
grid1 = grid(8,10, 10,10, 300,200, wg);
```

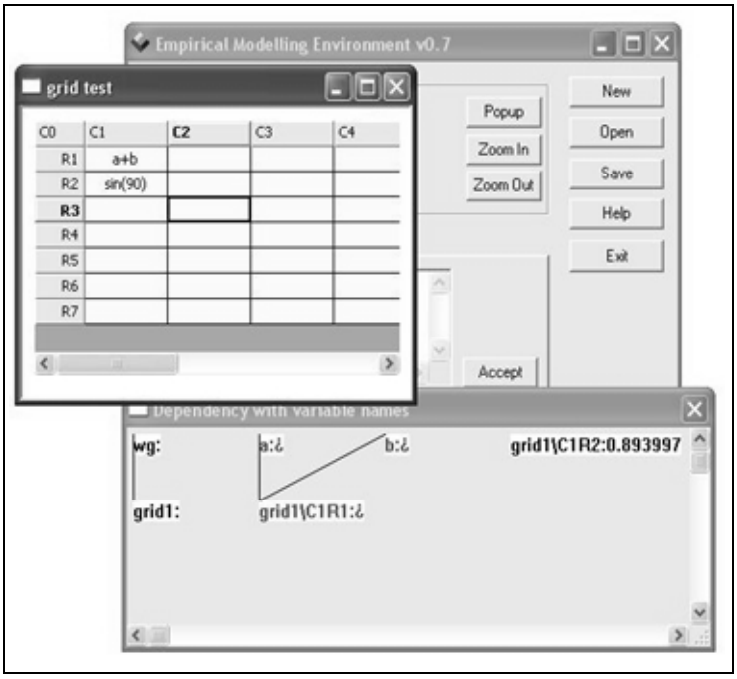

Figure 7.11: A built-in spreadsheet widget

The modeller can enter definitions into the cells of such a grid. For example, we have entered a definition `a+b` in the location of column 1 and row 1. The actual definition created is `grid1\C1R1 = a+b;`. The dependencies involved are shown in the model visualisation window at the bottom of the screen capture.

The spreadsheet grid illustrated in Figure 7.11 is actually an ActiveX control object obtained from the Web. This has demonstrated that we can link interface widgets created by using other programming language to an EM tool to facilitate the modelling of software systems with graphical user interfaces (cf. the interactor widgets in Penguims [Hud94]).

7.5.5 Evaluation

Throughout the development of EME, we have experimented with several new ideas that relate to improving interfaces to EM tools. In this subsection we give a brief evaluation of each of the experimental features we have introduced into EME:

- The *graphical visualisation* of an EM model can give an overview of all observables and dependencies within the model. This can help the modeller to comprehend the model. However, the visualisation supplied by EME is still

not satisfactory. For example, the layout of the symbols at each level of the hierarchy is random. When there are a lot of dependencies between two levels, edges cross each other so that it is very difficult to see which symbol is dependent on which symbol. There is no mechanism in EME to rearrange the symbols to reduce edge crossings.

- The new technique of *variable referencing* has the potential to increase the expressiveness of the definitive script whilst at the same simplifying its syntax. However, more experiments need to be conducted to justify these claims.

- *Procedural macros* provide a simple way to automate some repetitive interactions with the model. At this stage, there are only two types of macro. We need to design more macros to accommodate a variety of user interactions. Examples are copy and paste macros and file manipulation macros.

- We have explored the possibility of introducing *complex interface widgets* into an EM tool. Further investigation is needed to justify whether this is a suitable method for extending the tool. It would be even more useful if we could design interface widgets using primitive drawing definitions (cf. the design of the Dishwasher interface in chapter 3). This gives the user much more definitive control over the functionality and appearance of an interface.

7.6 Prospects

The research described in this chapter suggests that it is difficult if not impossible to build an EM model by using OO languages or pure functional languages. Neither paradigm can fulfil the demands of supporting essential characteristics of the DMF. In particular, the concept of EM agents is neither supported by OO nor functional paradigms. In fact, the DMF has a very different philosophy from OO or functional paradigms – the OO paradigm advocates data encapsulation but the DMF advocates free access to data; the functional paradigm rejects procedural access to variables but the DMF includes procedural access (actions) to variables as essential to the representation of agents.

We have found that the degree of conceptual and interface support for EM that Excel with macros supplies is similar to that supplied by TkEden. However, TkEden is still a better choice than Excel. The main reason is that it supports a variety of

different special-purpose definitive notations. Every definitive notation extends TkEden to support many new data types and different ways of specifying definitions. For example, by using Donald and Scout we can build definitive visualisations that cannot be easily built by using Excel. From a research perspective, since we can access the source code of TkEden, we can always incorporate and rebuild TkEden to reflect new ideas.

Where conceptual support for EM is concerned, the main limitation of existing EM tools is the lack of a mechanism to specify truly concurrent agency. Although pseudo-concurrency is possible by including a clock agent in TkEden, we need to incorporate real concurrency in order to study the full potential of the DMF in practice.

Where the interface for EM is concerned, we need more flexible ways to organise definitions and actions into agents. The development of WING and EME has helped us to explore two new ways to organise definitions and actions – namely directory-like organisation and syntactical organisation of agents. In the next chapter, we shall introduce another way to organise definitions that is arguably better than these two ways.

Another possible enhancement to existing EM tools is to provide a mechanism to generate an LSD specification automatically from a model. This idea was inspired by our experience of extracting an LSD specification from the Dishwasher model, as described in chapter 3. This experience suggests the following general informal rules for building each part of an LSD specification for an agent from a simple definitive script with actions:

- derivates – include all definitions in the agent as derivates.
- states – include all definitive variables on the LHS of derivates in the agent as states.
- handles – include all definitive variables on the LHS within the agent's actions as handles.
- oracles – include all definitive variables on the RHS of derivates as oracles.
- protocols – search for `if` statements in the procedures. The logical expression used by an `if` statement is the LHS (guard) of a protocol, and the sequence of definitions within the `if` statement comprise the RHS of the protocol. Any sequences of definitions that are not guarded by `if` statements in a procedure belong to a protocol with a `true` guard.

One requirement for generating LSD automatically from a script is that TkEden should ‘know’ about (i.e. have an internal representation of) agents. However, this is not the case in the current implementation of TkEden. In the future, if TkEden can be enhanced to include information about agents, further research into the automatic generation of LSD specifications from TkEden models would be viable.

7.7 Summary

In this chapter, we have discussed the evaluation of EM tools from two perspectives. With reference to conceptual support for EM, we have considered the directness of the translation from DMF models to particular implementations. At the tools interface, we have explored techniques that make the process of EM easier to perform. We have investigated three existing technologies, Java, Excel and Forms/3, as possible EM tool implementations. From this we concluded that it is difficult to use OO or pure functional languages to perform the activities of EM. We have also found that model-building in a spreadsheet with event macros captures most of the essential characteristics of the DMF. Both spreadsheets and TkEden reflect a major limitation of current tools for EM: their lack of support for the representation of agents with concurrent actions. We have also described some interface issues for TkEden. The developments of WING and EME explore new ideas that address problematic issues of TkEden. Finally, we have highlighted some major prospects for tool development in the future.