8 The Dependency Modelling Tool

In this chapter, we shall describe a new EM tool, the Dependency Modeling Tool (DMT). The motivation for developing the DMT is similar to that of developing WING and EME – they aim to enhance users' experience in the process of EM. However, the emphasis in DMT is on the ways to visualise various structures that commonly exist in EM models.

We start our discussions by identifying these structures in section 8.1. In section 8.2, we describe some tools developed by others in relation to visualising structures that are similar to the ones that exist in EM models. The development of DMT is partly inspired by some features of these tools. In section 8.3, we introduce DMT's user interface with a simple example. Two major considerations for the development of DMT are model comprehension and reuse. In section 8.4, we discuss how DMT facilitates model comprehension. In section 8.5 we shall discuss how DMT facilitates model reuse. The final section highlights various issues related further research and development DMT.

8.1 Structures in an EM model

The term 'structure' is used in this chapter to refer to some recognised pattern associated with an EM model. These patterns are directly related to the understanding of the model with respect to its context which is gained from the modelling process. For example, the definition of "`a is b+c;`" has the structure of dependency: the value of observable `a` is dependent on the values of observables `b` and `c`. Dependency is not the only type of structure that exists in an EM model. In fact, there are three common structures that can be easily distinguished from a script: *dependency structure*, *locational structure* and *contextual structure*. Dependency structure is the pattern of which observables are related to each other; locational structure refers to the physical organisation and arrangement of definitions in a script; contextual structure to grouping of definitions according to different contexts

for observation and interpretation. In our experience, it is usually necessary for the modeller to keep all the structures in mind during the modelling process for purposes of model comprehension.

In a TkEden script, the dependency structure is determined by formulae of definitions. Locational structure is determined by the organisation of definitions in this list (see left-hand-side of the Figure 8.1). There is no direct support for representing contextual structure.

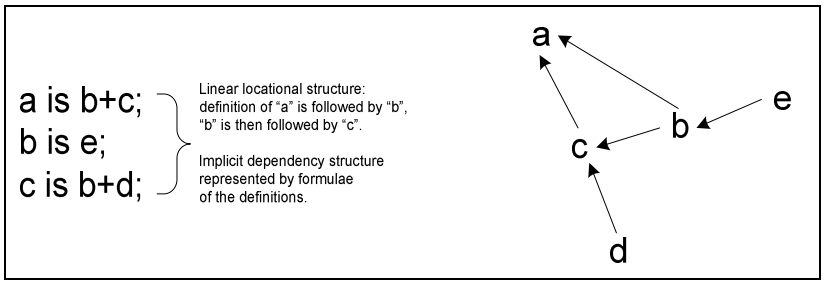


Figure 8.1: Linear locational structure and implicit dependency structure represented by a script of three definitions (left) and a dependency structure graph of the same definitions (right).

Abstractly, we can represent a dependency structure by a directed acyclic graph[4] (showing all dependencies among observables in the model). In the graph, observables are represented as nodes, and dependencies as edges. We can lay out the graph hierarchically: the nodes at the higher levels are dependent on the nodes at the lower levels of the graph[5]. Therefore, nodes at the lowest level of the graph are constant observables or 'undefined observables' (see right-hand-side of the Figure 8.1).

The importance of contextual structure seems to have been largely overlooked in our previous work, although there have been some attempts to deal with them implicitly (e.g. via openshapes in Donald [Bey86] and virtual agents in DTkEden [Sun98]). In Donald, we can define an openshape whose shape is determined by a set of other shapes. For example, an openshape `S` with two lines `L1` and `L2` is defined as:

[4] Similar to a spreadsheet, in an EM model cyclic dependencies are not explicitly represented. This is because cyclic dependencies cause an infinite loop of variable updates.

[5] This hierarchy is the basis for determining the order of variable updates. For example, a topological sort can be performed based on the hierarchy, which can minimise the number of evaluations required for variable updates. Synchronous updates are also possible while still maintaining the integrity of the model.

```
%donald
openshape S
within S {
  line L1
  line L2
}
```

Subsequently, we can refer to two lines individually by `S/L1` and `S/L2`.The contextual structure in this case is accommodated by a syntactic construct of within-clause and a reference symbol '/'. In DTkEden, we can associate definitions with a virtual agent. The following is an example of virtual agent declaration:

```
>>bird
windspeed is 20;
height is 1000;
>>
```

This defines a virtual agent `bird` with two observables. The symbol '`>>`' at the beginning and the end of the declaration specifies contextual information – in this case `windspeed` and `height` belong to the `bird` agent. The actual definitions created by the above declarations are:

```
bird_windspeed is 20;
bird_height is 1000;
```

Literally, a prefix '`bird`' has been added to both definitions with a separator '_'.

Representing dependency, locational and contextual structures by using textual syntax in TkEden and DTkEden has a major limitation: it is difficult for a modeller to understand these structures in isolation from other syntactic constructs. Two new tools introduced in the previous chapter had made attempts to address the limitations – WING provides direct support for organising the contextual structure by visualising using a tree explorer similar to the file explorer and locational structure by spreadsheet-like cells. EME visualises the dependency structure by drawing a dependency structure graph. But the results are still not satisfactory.

The aim of the research described in this chapter is accordingly to find better ways of representing the structures that are common to all EM models. On this account, we have developed DMT to represent the structures graphically. We believe that by representing the structures graphically in a coherent way, the experience of *building an EM model as an artefact* can be significantly enhanced. At the same time, the research enhances the prospects of making EM tools more accessible and usable for general users.

8.2 Inspiration from other tools

In this section, we discuss some existing tools that organise structures similar to the structures in EM models. Unlike TkEden script, these tools use graphical techniques to organise the structures. The development of DMT was partly inspired by our experience of using these tools.

A common contextual structure can be found in modern operating systems. That is the organisation of files by a hierarchical structure of directories. Interfaces like the file explorer provide a graphical representation of the directory structure. Most PC users nowadays use them to manage their files instead of typing in command prompts. There are some limitations on using a hierarchical structure to represent contextual structure in EM model. We shall discuss them later in this chapter. However, the idea of organising files by an explicit representation of contextual structure is invaluable to the usability of modern computers.

The importance of explicitly representing both locational and contextual structures is well attested by a popular note taking thinking skill called Mind Mapping [Buz95]. Figure 8.2 shows a Mind Map about the contents of this chapter. A Mind Map is a hierarchical graph with the highest level root located in the centre and branches radiating out in all directions. The root represents a central context of interest. The branches with keywords written on them represent concepts in the context of the keyword from a higher level branch. Relative locations between branches can also convey meanings. Empirical studies of Mind Map use indicate that identifying and managing the hierarchical structures associated with a concept helps people to organise and think about the concept more naturally and creatively [Buz95].

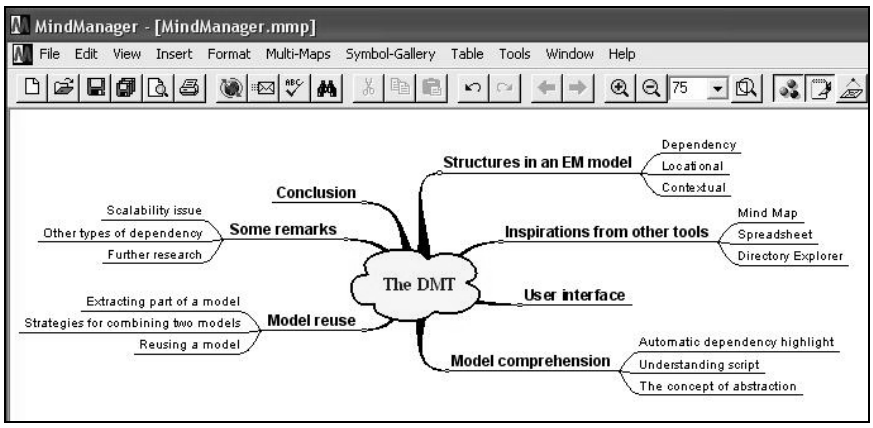


Figure 8.2: A Mind Map about the contents of this chapter created by using MindManager.

The Mind Map in Figure 8.2 was created by using the MindManager tool [Min00]. One important feature of the tool is that it allows the user to move the nodes freely. This can be interpreted as allowing the user to change the locational structure of the Mind Map. It helps users to organise information according to their subjective preference and, therefore, has cognitive significance.

The feature of explicitly representing dependency structure can also be found in connection with understanding a spreadsheet. The dependencies between cells in a spreadsheet are normally hidden from the user. This makes a spreadsheet difficult to understand [Gre98a]. Newer versions of spreadsheet applications contain a dependency tracing feature. For example, Excel can trace dependencies between cells by showing arrows, as shown in Figure 8.3.

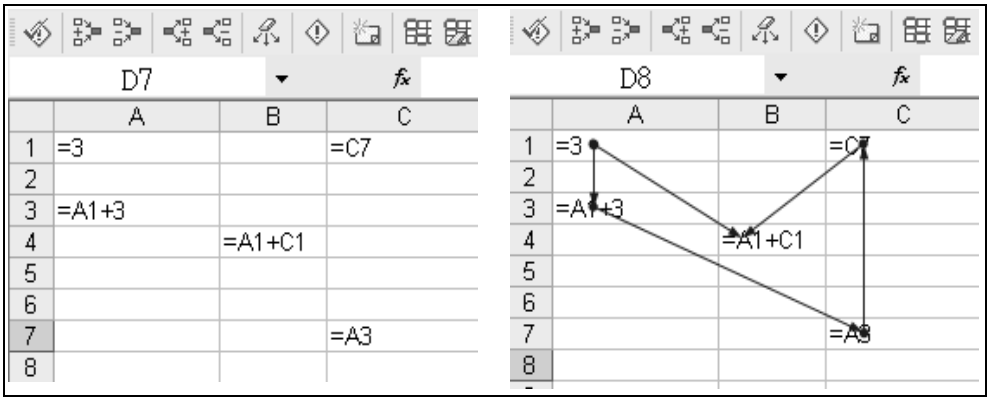


Figure 8.3: A spreadsheet in Excel (left) and its dependency traces (right).

8.3 User interface

The development of DMT is motivated by the need to enhance users' experience of the process of EM. DMT provides features for users to build EM models as artefacts that are *visually* as well as *physically* more tangible than a definitive script – it uses acyclic graphs to visualise three common structures (dependency, locational and contextual structures) that exist in an EM model, and provides means to manipulate them directly by using a pointing device. In addition, a user can extract definitions created by DMT as Eden definitions, or conversely import Eden definitions from a definitive script. The current version of DMT is implemented in Java with standard Java libraries, so it is platform independent. Figure 8.4 shows the user interface of DMT with an empty model.

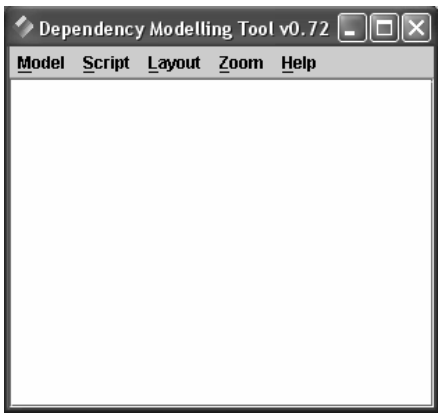


Figure 8.4: User interface of DMT

The interface provides a large empty white area for drawing the dependency graph structure of an EM model. The major functionalities of DMT are reflected by the primary menu options – *Model* provides save, load, print and combine models functions; *Script* provides translation functions to and from definitive scripts; *Layout* provides functions to automatically arrange graphical positions of the nodes; *Zoom* provides functions to scale the entire graph; *Help* provides online help for using the interface (see Appendix H for menu reference).

The basic means of entering a definition can be explained by creating a simple definition: `a is b+c;`. Figure 8.5 shows a sequence of steps to create a graph of this definition and the mechanism to move around the nodes of the graph. The figure illustrates the following basic features of DMT:

- A node can be created by clicking the right mouse button.
- The definition of a node can be edited by double-clicking the node with the left mouse button.
- Any undefined observables will be automatically created as new nodes.
- The details of a node can be checked by pointing at it with the mouse. The details are shown at the top-left region of the graph.
- A group of nodes can be selected by drawing a rectangular selection box around them.
- The selected group of nodes can be moved by drag and drop manipulation of the rectangular selection box (individual node can also be moved by drag and drop without a selection box).

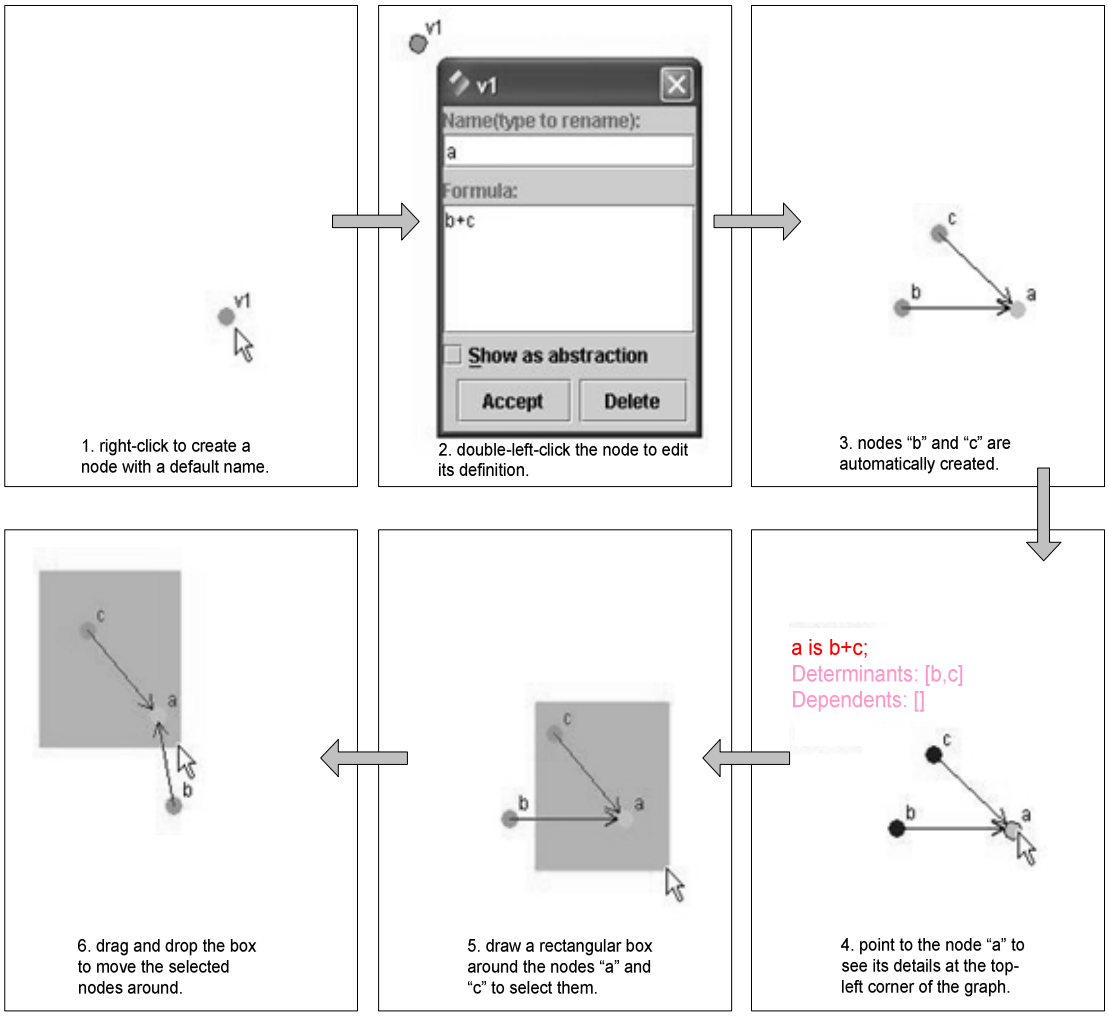


Figure 8.5: A sequence of steps to enter a definition and to move nodes around.

DMT uses a *colour coding* for different graphical elements of the graph. Unfortunately, the figures here are printed in black and white. However, the colour coding has significance in understanding the graph. Examples of the colour coding are: nodes with definitions are coloured in grey; nodes with no definitions are coloured in green; the selection box is in light blue.

The semantics of a DMT model, when interpreted as an EM model, can be summarised as follows:

- Observables are nodes.
- Dependency structure is represented by directed-edges joining the nodes. For example, if node `a` depends on node `b`, there is a directed-edge pointing from `b` to `a`.

- Locational structure is represented by arrangement of nodes in a two-dimensional space.
- Contextual structure is represented by abstractions (which will be introduced in another section).

By providing features to graphically represent the structures in an EM and means of directly manipulate them, DMT improves the comprehensibility of the model. In addition, it provides mechanisms to allow easy reuse of existing models. We shall discuss model comprehension and reuse in more detail in the following two sections.

8.4 Model comprehension

DMT provides various features to help the user gain and maintain understanding of the developing model in the process of EM. We shall discuss these features under three headings: automatic dependency highlighting, understanding scripts and abstraction.

8.4.1 Automatic dependency highlighting

As mentioned before DMT uses colour coding to help the user understand the model. For instance, the dependencies related to a particular node are highlighted automatically. By way of illustration, the left-hand-side of Figure 8.6 shows a graph associated with the script of five definitions:

```
a is b+c;
b is 10;
d is a;
c is 10;
e is a;
```

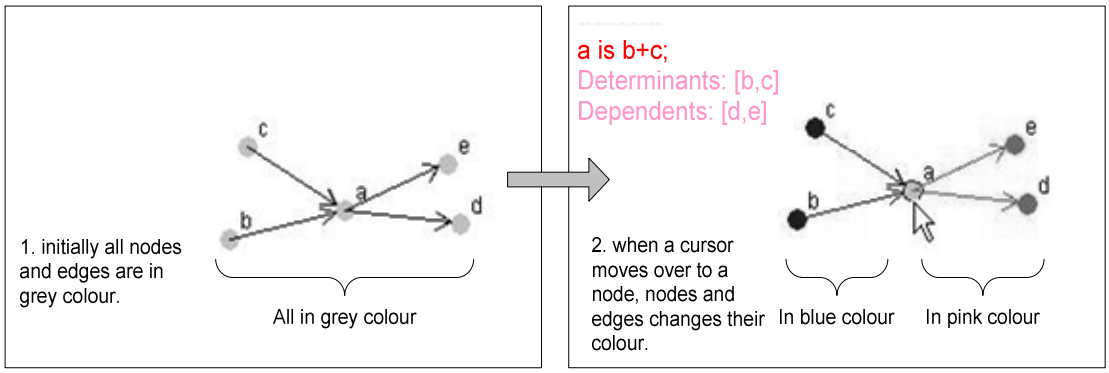


Figure 8.6: Automatic dependency highlighting

Initially all the nodes and edges are in grey colour. When the user moves the cursor over node `a`, DMT immediately highlights the nodes and edges associated with its determinants in blue and its dependents in pink (see right-hand-side of Figure 8.6). This automatic visual feedback feature is very useful especially when studying a model with a large number of nodes and edges.

8.4.2 Understanding scripts

We can use DMT as a tool for understanding existing models represented by definitive scripts. DMT can import a definitive script, interpret it and find out all the observables and dependencies represented in it. Since the only positional information explicit in the script is the linear order of the definitions, we need some methods to lay out the graph. There has been much research on algorithms for the automatic arrangement of directed graphs (e.g. [Sug81, Tol96, Pur00]). A typical criterion used for arranging the nodes is minimizing edge crossings. Ordering a directed graph hierarchically is also common. We found that such strategies are of limited use for arranging the layout of an EM model.

The geometric location of nodes in a DMT graph conveys information about a modeller's understanding of the model. A modeller's subjective perspective on the model, as reflected by the location of nodes, is difficult to capture in automatic layout algorithms. Our experience shows that one of the most effective ways to use the DMT is to allow the modeller to arrange the position of the nodes manually. For example, Listing 9.1 shows a definitive script of an ATM model. This script is imported into DMT by choosing the *Script* and *Direct Import* menu options.

```
1.  %eden
2.  userInput is [PINentered,required];
3.  currtime is 71;
4.  required is 0;
5.  overLimitToday is accLimitPerDay <
    (accDrawnToday+required);
6.  accLimitPerDay is 200;
7.  transpossible is cardInMachine &&
    cardValid &&  bankValid  && PINvalid
    && !overLimitToday && idValid &&
    accStatus && !accOverLimit &&
    moneyReady;
8.  accDrawnToday is 0;
9.  ATMbank is ['A', 'B', 'C'];
10. accStatus is 1;
11. environment is
    [currtime,cardInMachine];
12. cardInMachine is 0;
13. cardExpiryDate is 320;
14. r10 is (required - 20*actual20) / 10;
15. card is [cardBank, cardID, cardPIN,
    cardStartDate, cardExpiryDate,
    cardStatus];
16. ATMtens is 100;
17. r20 is required / 20;
18. accOverDraftLimit is 10;
19. moneyOut is
    [transpossible,actual5,actual10,actua
    l20];
20. actual10 is
    (ATMtens>=r10)?r10:r10-ATMtens;
21. cardStartDate is 1;
22. cardValid is (cardStatus==1) &&
    (currtime >= cardStartDate) &&
    (currtime <= cardExpiryDate);
23. cardBank is 'A';
24. actual20 is
    (ATMtwenties>=r20)?r20:r20-ATMtwe
    nties;
25. cardID is 123;
26. ATMtwenties is 100;
27. ATMfives is 100;
28. PINvalid is cardPIN == PINentered;
29. actual5 is
    (ATMfives>=r5)?r5:r5-ATMfives;
30. accOverLimit is required >
    (accTotal+accOverDraftLimit);
31. cardPIN is 999;
32. accTotal is 10000;
33. ATMcardIDlist is [123, 321];
34. r5 is (required - 20*actual20 -
    10*actual10) / 5;
35. moneyReady is
    (actual5*5+actual10*10+actual20*2
    0)==required;
36. PINentered is 999;
37. cardStatus is 1;
38. ATMbalance is ATMfives *5 +
    ATMtens*10 + ATMtwenties*20;
39. idValid is isin(cardID,
    ATMcardIDlist);
40. accDetails is [accStatus,
    accLimitPerDay, accTotal,
    accDrawnToday,accOverDraftLimit];
41. bankValid is isin(cardBank,
    ATMbank);
```

Listing 9.1: A definitive script of an ATM model

After importing the script, DMT randomly positions all the nodes representing observables in the script. The result is usually a graph with a messy arrangement of nodes where many edges cross over, and it is difficult to understand (see Figure 8.7). However, the modeller can get more understanding of the model by moving around the nodes interactively using a pointing device. Moving a node around immediately contributes to the understanding of the determinants and dependents of the observable that the node represents (because of the feature of automatic dependency highlighting). Further grouping of the nodes assists in gaining a better understanding of the model. Eventually, as shown in Figure 8.8, a well-organised layout that reflects the semantics of the model will typically emerge.

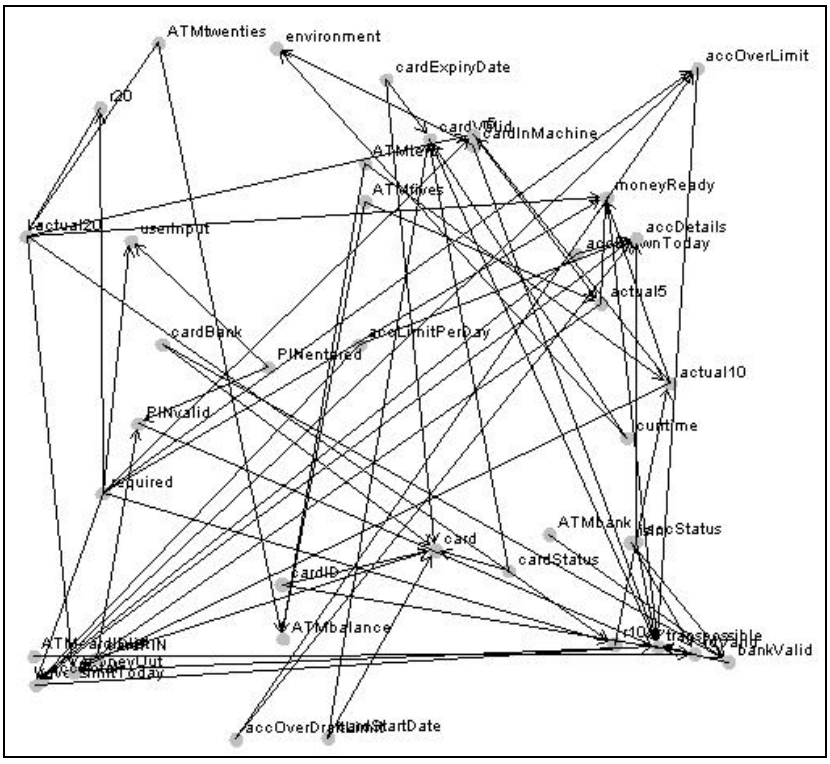


Figure 8.7: Random layout of the ATM dependency graph

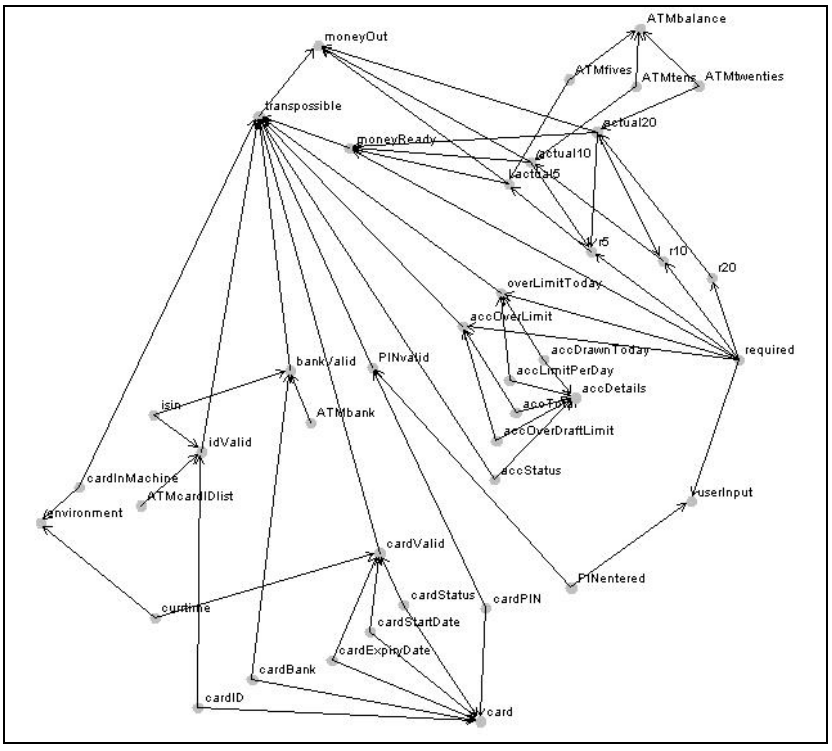


Figure 8.8: Organised representation of the ATM dependency graph

For a small model, a modeller can rapidly understand the model. However, when the model is larger, and consists of say 100 nodes and 300 dependencies, moving the

nodes around becomes tiresome. The exploration of the model becomes difficult. This can be solved if we have knowledge of some observables that are more important than the others. If a modeller knows the key observables, nodes can be more easily arranged by firstly locating the nodes corresponding to the key observables, then the observables that are directly connected to the key observable and so on. In the ATM case, if we know that `transpossible` is the most important observable, the arrangement of nodes can be based on it.

For any one particular definitive script, there is a virtually infinite number of ways to layout its dependency graph. Different modellers end up with different layouts even if they all start from the same random layout. This in part reflects the fact that we all understand a particular concept differently. Building a model by arranging the nodes can contribute directly to our construal of the model. The geometric positions of the observables embody part of our understanding of the model.

Apart from understanding an existing model, geometric positioning of nodes can also help in building new models. In this case, the modeller positions an observable (node) each time he or she introduces a definition. Grouping observables and moving groups of observables in conjunction with the model-building activity can contribute visual support for model understanding as it evolves.

8.4.3 Abstraction

This subsection explains the concept of ‘abstraction’ in DMT. The contextual structure of the script can be represented in a way that is similar to the directory navigation of files in a modern operating system. Johnson et al. [Joh99] discuss different ways of representing a directory structure such as outline views, tree diagrams, Venn diagrams and tree-maps (see Figure 8.9).

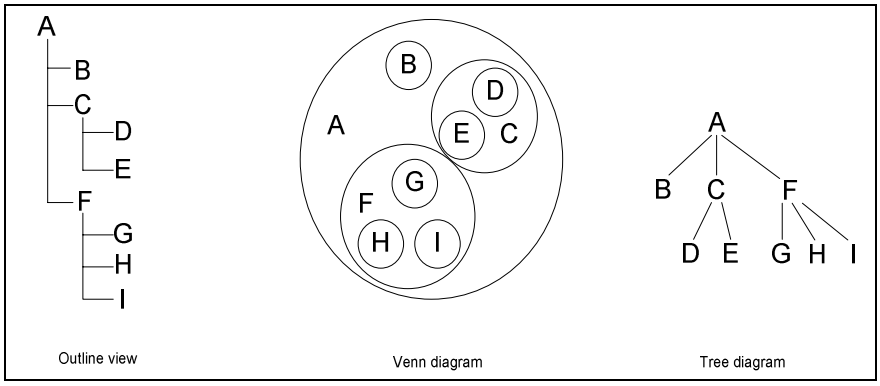


Figure 8.9: Different representations of directory structure

The design of the WING interface attempts to mimic outline view navigation (see Figure 8.10). A user can create new containers that contain sets of definitions just as directories contain sets of files. A dependency is implicitly defined in the sense that a container is dependent on the aggregation of definitions that it contains.

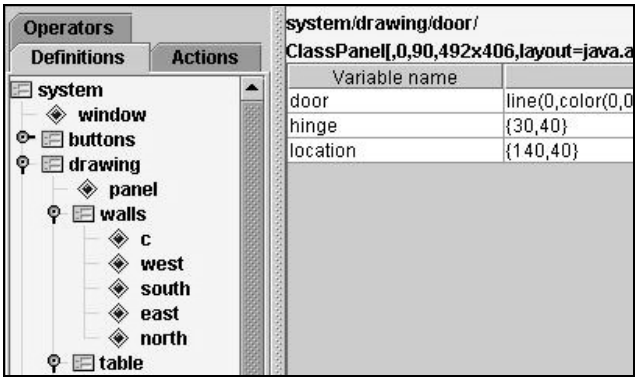


Figure 8.10: Outline view of containers in WING

An outline view, however, cannot represent a node with two parent nodes. A typical topological tree of an EM model has nodes with one or more parents. For example, consider the status of the variable c in following definitions:

```
a is b+c;
e is c;
b is 10;
c is 10;
```

There is no direct way of representing the dependency using an outline view. Only the other two kinds of directory representation can be used, as shown in Figure 8.

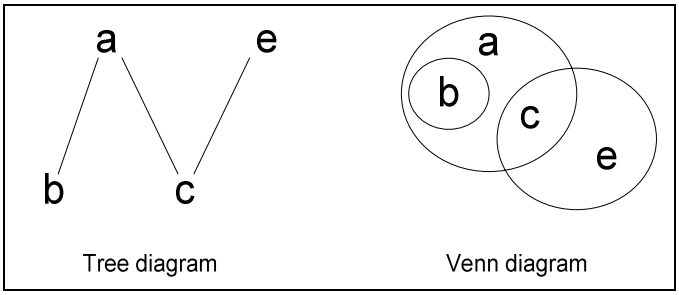
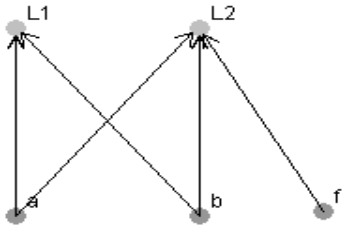


Figure 8.11: Diagram with a node with 2 parents and the Venn diagram.

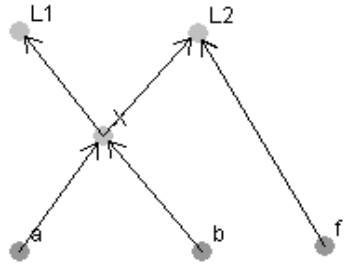
Abstraction in DMT combines the merits of the tree and Venn diagram. We can understand abstraction by firstly consider two example definitions:

```
L1 is (a+b)*4;
L2 is f+a+b;
```



By examining the formulae in these two definitions, it is obvious that they are both dependent on observables `a` and `b` or more precisely on the expression `a+b`. This knowledge of pattern can be captured by replacing the two definitions by the following three 'equivalent' definitions:

```
L1 is X*4;
L2 is f+X;
X is a+b;
```



The third definition here is an abstraction of what we observed. Observable `X` is at a higher level of abstraction than the other observables. To represent the fact that `X` has an abstraction level different from the others, DMT allows a modeller to visual `X` differently by directly specifying it is an abstraction. If `X` is an Abstraction, the colour of it becomes orange and there is a round-cornered orange rectangle that embraces `X` and all its determinants. Figure 8.12 shows a sequence of steps to specify `X` as an abstraction. This figure also shows that the edges from `a` and `b` are hidden as a result of declaring `X` as an abstraction.

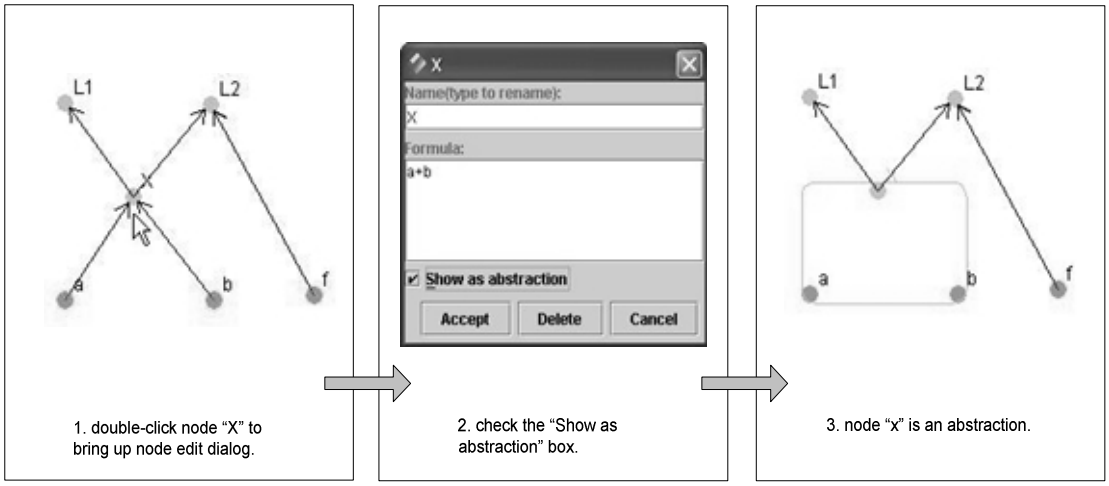


Figure 8.12: A sequence of steps to set up an abstraction.

Defining an abstraction can also be viewed as a way of hiding excessive complexity. For a large model, hiding edges directed from the determinants can make the graph less messy. For example, Figure 8.13 shows an observable `Z` that depends on another 20 observables. Specifying `Z` as an abstraction hides all the edges directed towards it.

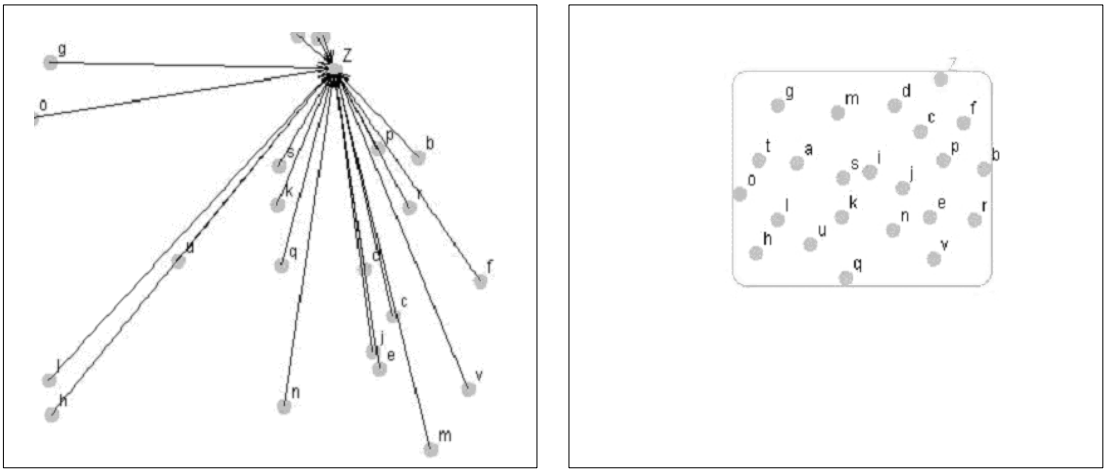


Figure 8.13: Two different representations of the same model – normal representation (left) and 'Z' as an abstraction (right).

Defining an abstraction is also a way to explore agency. In the ATM model, we can specify the observables `card` and `cardValid` as agents. As shown in Figure 8.14, their abstractions overlap each other. This might give a clue to the modeller that two separate sets of `card` and `cardValid` observables are needed.

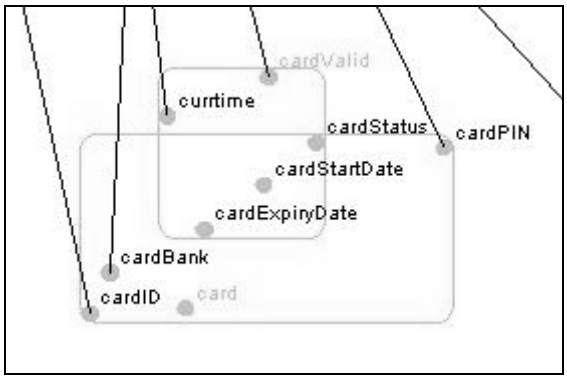


Figure 8.14: Overlapping of two abstractions.

Abstraction can be a unified way of representing agents, directories in Jam2 [Car99], structures in Modd [Geh98] and containers in WING. To summarise, abstraction can be used for:

- Hiding excessive information
- Deriving agency
- Representing agency

By experiencing this way of identifying agents, we notice that an LSD specification can be viewed as a result of the modelling process but does *not* have the generality of an arbitrary script-based EM model. This is because an LSD specification is more suitable for representing settled agents. It does not have the degree of openness that a DMT model has.

Other kinds of abstraction may also be usefully introduced into the DMT. A counterpart of Harel's hierarchical organisation of states in a statechart [Har88] is one possible candidate. As will be illustrated later in connection with modelling a draughts game, it would be useful in some contexts to be able to abstract groups of observables that exhibit a generic dependency pattern (cf. the observables relating to a single square of the draughts board, as displayed in Figure 8.21.).

8.5 Model reuse

Apart from model comprehension, the other main contribution of DMT is new ways of reusing an EM model. In subsection 8.5.1, we shall describe a mechanism to extract Eden definitions from part of a DMT model. This mechanism is very useful for selecting reusable parts of a model. Model reuse in DMT is based on

well-defined strategies of combining two EM models. We shall discuss the strategies in the subsection 8.5.2 and give a simple example of model reuse in subsection 8.5.3.

8.5.1 Extracting part of a model

It is common for some part of an EM model to be generic enough to be reused in building a new model. DMT allows a modeller to extract part of a model and save it for later reuse. Figure 8.15 shows how to extract part of an existing model into a definitive script. DMT automatically appends `%eden` at the beginning of the extracted script. The modeller can save the extracted script to a file for later reuse.

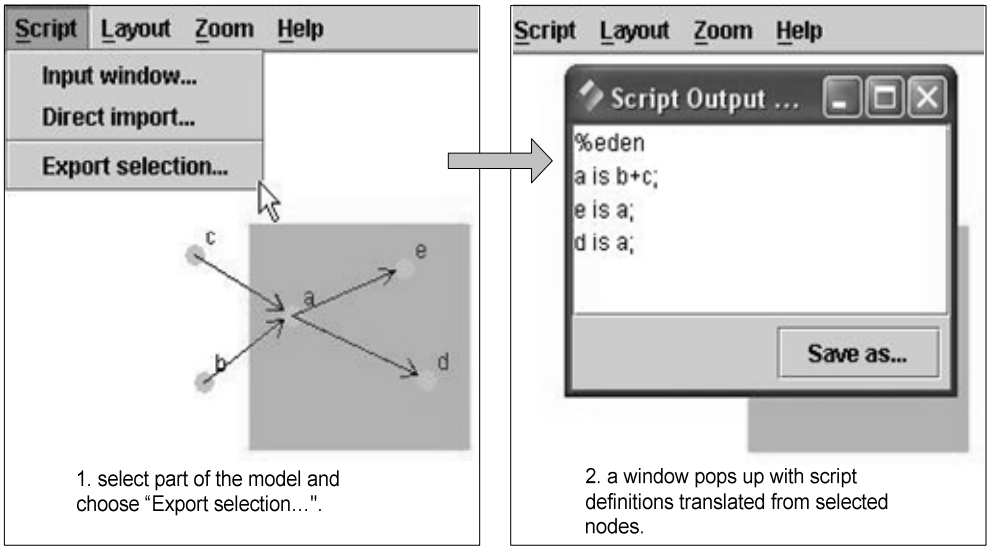


Figure 8.15: Extracting script definitions from a DMT model.

8.5.2 Strategies for combining two models

A definition has three ingredients: the definitive variable (or 'observable') at the left-hand-side of the definition, the formula at the right-hand-side and the current value of the variable. What does it mean to say that the definition of a variable is well-defined? Does it mean that all three ingredients of the definition are defined? – or that some ingredients are defined and some are not? We have to take a closer look at each individual ingredient of a definition before answering these questions.

A definitive variable is a metaphorical representation of some external observable. This means in effect that a variable is 'defined' as soon as it is referenced by any definition. That is to say, a particular variable should be treated as defined not

only when it is given a defining formula, but when other definitions refer to it in their formulae. Only when a variable is defined in this sense can other ingredients of its definition become meaningful. If a variable has a defining formula, its value can be determined by attempting to evaluate the formula, and if the evaluation is successful, we have a defined value for the variable. On the other hand, if the evaluation fails, the value for the variable will be undefined.

In modelling a situation, the modeller may initially have only a vague idea of what defining formula is appropriate when he or she decides to introduce a variable. In that case, although the variable is defined, its formula is not yet defined and neither is its value. The following table (see Figure 8.16) illustrates all the possibilities that can arise when a variable has been first defined.

	Formula defined	Formula undefined
Value defined	A defined definition	Impossible case
Value undefined	An evaluation exhibits an error.	The dependency of the variable is still subject to investigation.

Figure 8.16: Cases when an observable is defined

DMT's strategies for combining existing models are based on the above notions of defined and undefined ingredients. Here is an example. Suppose we have two models `X` and `Y`. We want to combine them to form a model `Z`. This can be written as:

```
Z = X union Y
```

The general rule for combining two models is *to preserve as much knowledge about observables within the two models as possible, subject to avoiding conflict*. For example, if the first model has observable `v` defined and the second has not, the resulting model will have an observable `v` defined as it is in the first model. Figure 8.17 shows possible cases relating to the definition of `v` in combining `X` and `Y` to form `Z`.

model	case 1	case 2	case 3	case 4	case 5	case 6	case 7	case 8	case 9
X	@	@	@	v is 20	v is 20	v is 20	v is @	v is @	v is @
Y	@	v is 10	v is @	@	v is 10	v is @	@	v is 10	v is @
Z	@	v is 10	v is @	v is 20	conflict	v is 20	v is @	v is 10	v is @

Figure 8.17: An example of possible cases for combining models X and Y to form Z ('@' means 'undefined').

The problematic case is case 5, where definitions of `v` exist in both `X` and `Y` with

different formulae. This conflict cannot in general be resolved by applying rules automatically. For example, if we are currently building a model `X` and we want to reuse model `Y` as a building block in `X`, we should choose which definition we actually want manually.

8.5.3 Reusing a model

Reusing a model can be interpreted as combining the model with the new model which we are currently building. Therefore, DMT uses the strategies discussed in the previous subsection to facilitate model reuse. It is convenient to explain and illustrate the idea by an example. The following script defines a generic triangle comprising three lines:

```
x1 is @;
y1 is @;
x2 is @;
y2 is @;
x3 is @;
y3 is @;
L1 is line(x1, y1, x2, y2);
L2 is line(x1, y1, x3, y3);
L3 is line(x2, y2, x3, y3);
```

Its DMT equivalent is shown on the left of the left-hand-side screen capture in Figure 8.18. Suppose our task is to define a generic pattern of two triangles sharing one vertex. In this case the shared vertex is `(x3, y3)`. The steps are as follows:

1. Save the generic triangle model into a file.
2. With the generic triangle still on screen, choose *Combine* to load the file. This brings up a window that contains a second generic triangle (left-hand-side of the Figure 8.18).
3. Rename the nodes in the newly loaded generic triangle to avoid name clashes with the existing ones except in the case of the vertex `(x3, y3)`.
4. Choose *Accept* to combine the two generic triangles into one figure. The result is shown in right-hand-side of Figure 8.18.

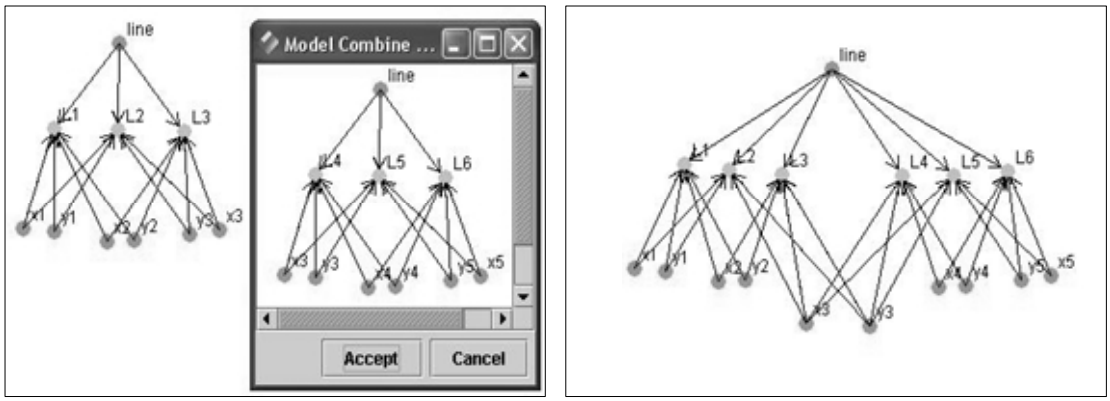


Figure 8.18: The Model Combine dialog with nodes renamed (left) and the result of combining two triangles (right).

As the example shows, when we want to reuse a model, we often need to rename the observables. The renaming of observables in a script is sometimes tedious. This is because the references to an observable may be scattered around everywhere in a long script. For example, we need to find and rename each of coordinates in the generic triangle script 3 times. With the graph representation, DMT centralises all references into one place. Therefore, in DMT, we need only carry out the renaming once for each coordinate to achieve the same result.

8.6 Some remarks

In this section, we discuss various issues related to further research and development of DMT.

8.6.1 Scalability issue

To test the scalability of DMT, we have tried to import many existing EM models in the form of definitive scripts. With a fair amount of time, we can generally rearrange the locations of nodes in each imported model from the initial random layout to a more comprehensible form. However, DMT has encountered problems when we try to visualise models with a large amount of dependencies. For example, the script for the board of an OXO game model contains 209 definitions and 814 dependencies (see Appendix I). After importing this model to DMT, we found that there is no way to rearrange the nodes to get a better layout out of the random layout (see Figure 8.19). In this case, DMT does give the user a hint about the complexity of the model.

However, it is difficult to go a step further in terms of understanding the model out of the random graph. With reference to Appendix I, we can see that the actual script is easier to understand! This illustrates a common scalability problem in visual language for modelling or programming that relates to the limited size of the screen display.

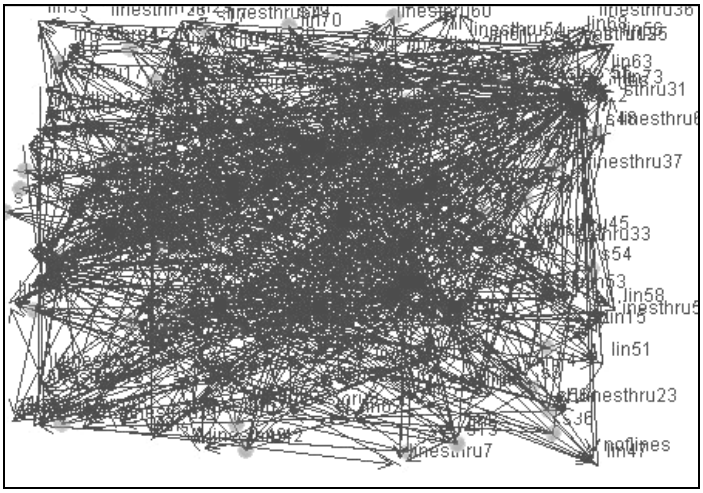


Figure 8.19: Visualisation of a definitive script with a large amount of dependencies

One possible solution is to allow user to select and visualise just parts of the whole complex model, and hide the remaining parts. A simple technique for extracting suitable subscripts for this purpose is to use a text editor to identify all definitive variables with a common pattern or feature. A more sophisticated technique, currently under development by EM research group involves storing the symbol table of a script in a relational database. All the definitions can be stored in a relational database implemented within TkEden using the Eddi definitive notation mentioned in section 2.3.1. The user can then use relational queries to select parts of the model as views. DMT can be developed to allow the user to link these views with their graphical representations. This technique has been used in studying a bug in an EM model described in the following subsection.

8.6.2 Potential for model debugging

One possible use of the DMT is to help the process of debugging EM models. By way of a practical example, we here study a bug in a draughts game model written using a TkEden script (see Figure 8.20). The draughts model contains an 8 by 8 board and some circular pieces. Each square on the board has a circle on it. The fill colour of the circle is as follows: if there is no piece occupying the square its colour should be the

same as the background colour of the square, otherwise, its colour should be black or white depending on the colour of the occupying piece. When the background colour of the squares was changed by assigning a new value to the observable `bgcol` the following problem resulted: if a piece was placed on a square and then removed, the fill colour of the circle no longer matched the background colour.

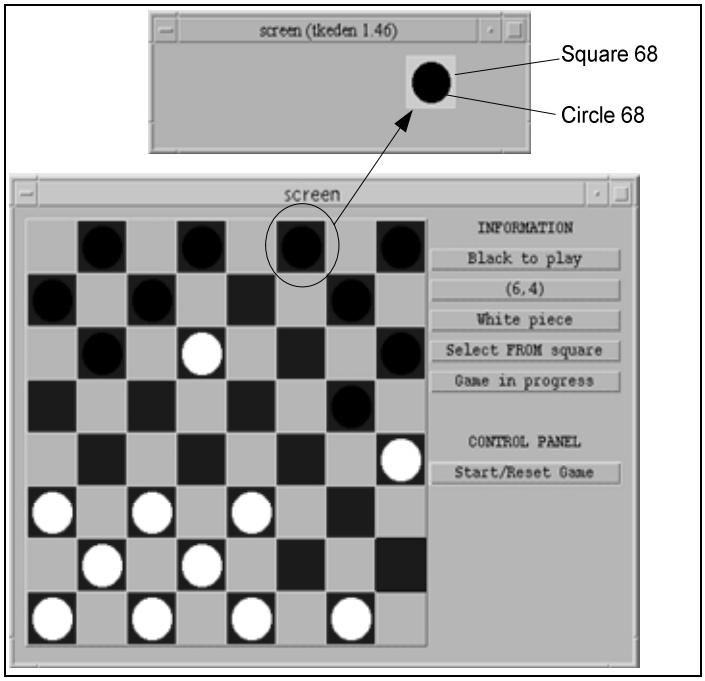


Figure 8.20: The draughts board (bottom) and the study of square 68 (top)

With the help of relation database queries such as we described in the last subsection, we are able to study the problem by selecting and extracting all the definitions relating to a particular square. In this case, we have extracted all the definitions relating to the square and circle on column 6 and row 8 of the board, as shown at the top of Figure 8.20 above. We then study the extracted definitions by using the DMT. Figure 8.21 shows the DMT of the definitions. After rearranging the nodes, we find that the DMT model divides into two sub-graphs: one for the definitions of the circle 68 and the other for the square 68 (see the top screen capture in the Figure 8.21). The fact that the fill colour of the circle (`bgcolor`) and the background colour of the square (`bgcol`) should be the same when no piece is on the square indicates that there should be a dependency between two colours. However, the DMT analysis tells us that there is no dependency between the circle and the square. The bug is removed by adding a new observable (`bckgrncol`) to represent their common colour and defining both `bgcol` and `bgcolor` to be equal to `bckgrncol`. In this way, we make a 'link' between the two separate dependency graphs, as shown at the bottom of Figure 8.21. By using the DMT, we found it easier

to understand the complex dependencies involved in the EM model. This illustrates one way in which the DMT can lead to more effective debugging of EM models.

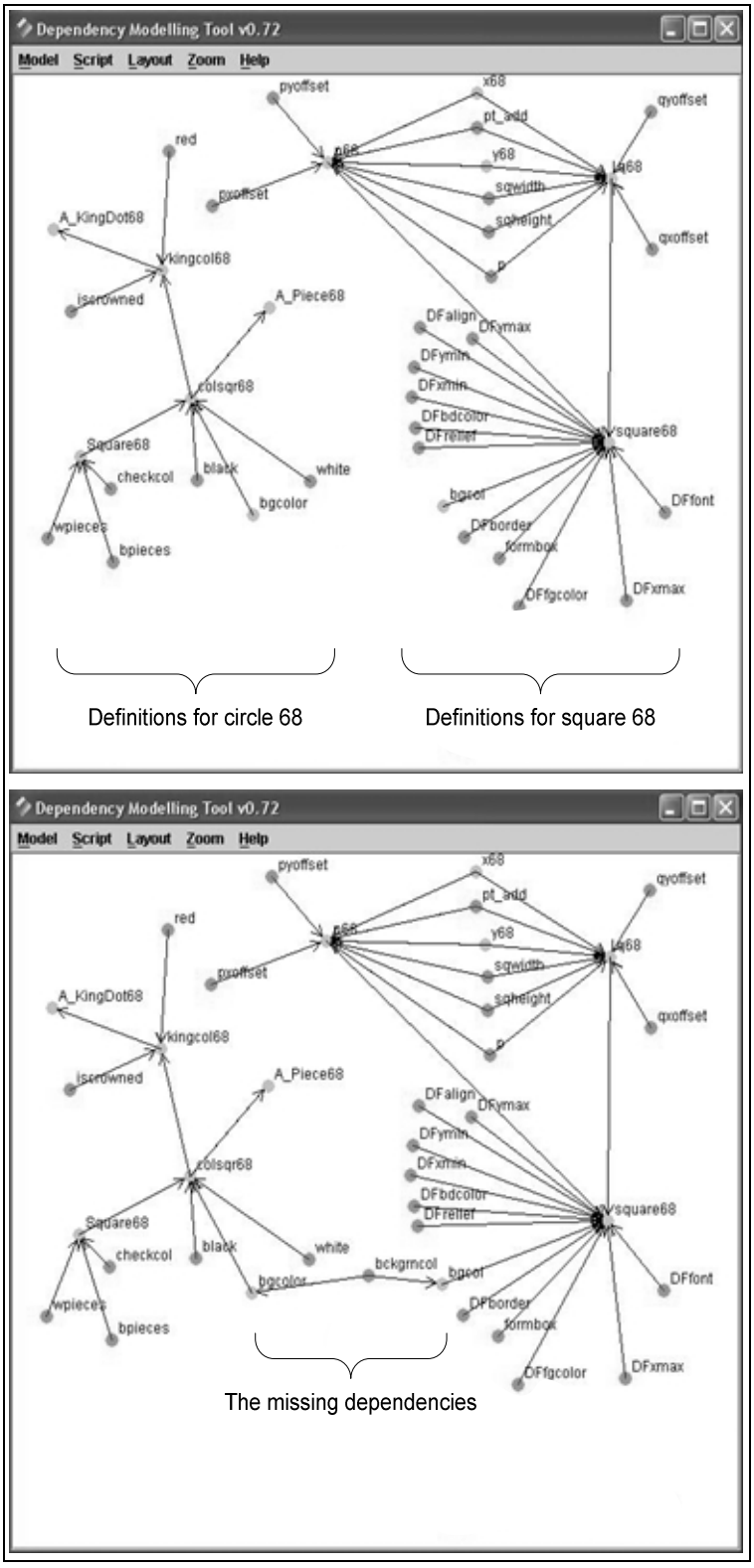


Figure 8.21: The DMT model for a single square of the draughts board (top) and the missing dependencies (bottom)

8.6.3 Other types of dependency

The development of DMT prompts us to ask a question: can we visualise all the dependencies that possibly exist in an EM model? This question leads us to identify different types of dependencies in EM models. In general, we can identify three types of dependencies: *definitive dependency*, *procedural dependency* and *dynamic dependency*. Definitive dependencies are specified explicitly by formula definitions. For example, the definition `a is b+c;` specifies a definitive dependency between observable `a` and its determinants `b` and `c`. As we have seen, DMT visualises this type of dependency by a directed graph. However, DMT does not visualise the other two types of dependency.

Procedural dependencies are implicitly established by actions. For example, the following action contains a procedural dependency:

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

This action monitors changes of `b` and `c` and *assigns* the sum of them to `a`. However, by just looking at this action, we cannot be sure `a` is merely dependent on `b` and `c`. This is because there may be other actions that also change the value of `a`. Only if we are sure that there is no other action that changes the value of `a` can we replace the action with a definition: `a is b+c;`. In this case, the procedural dependency is transformed to a definitive dependency. The transformation cannot be automatically established. This is because the fact that there is no other action that can change the value of `a` cannot be generated without intelligent intervention from the modeller.

Dynamic dependencies are also implicitly established by actions. But unlike a procedural dependency, a dynamic dependency involves actions making definitions. For example, the following actions establish a dynamic dependency for `a`:

```
proc x: v1{
    a is b+c;
}

proc y: v2{
    a is y;
}
```

In this case, the definition (not value!) of `a` depends on the changes of some other

external observables `v1` and `v2`. If `v1` changes, `a` will depend on `b` and `c`. On the other hand, if `v2` changes, `a` will depend on `y`.

In our experience, both procedural and dynamic dependencies are difficult to detect automatically. Detecting these dependencies may involve extensive analysis of the semantics of agents and actions in the model. On the other hand, to some extent they can also be seen as reflect the limitations of our current understanding of the scope for definitive dependency.

8.6.4 Further research

Apart from addressing the scalability issue and visualising other types of dependencies, there are many other possible interesting research topics and developments that can be conducted in relation to DMT in the future. Here we list some of them.

- Research on end-user interface – As we have mentioned in chapter 6, we can use an EM model to control a ubicomp system (as soft-interface). Techniques developed in DMT to visualise and manipulate an EM model are more user friendly than entering definitions using the input window of TkEden where small scripts are involved. Small scripts are arguably easier to build and understand by using the DMT approach than by direct use of TkEden.

- Developing other script translators – Currently, DMT allows only Eden definitions to be imported from a definitive script. Therefore, if 1t.4(6(r)8l(e)-6.110.a de)4.6rs,

TkTcl-based TkEden. Alternatively, TkEden can be used as a platform for implementing DMT.

- Development of a grid-free spreadsheet application – The main functionality of a spreadsheet application is not only calculation but also report generation. To our knowledge, all the commercial spreadsheet applications available at the time of writing this thesis are based on table layouts with grid reference. DMT provides an alternative grid-free layout. In this case, every node in the DMT graph represents a spreadsheet cell whose location can be arranged by the user freely. The user can arrange all the nodes into a report format for printing. For this purpose, the user can choose to print only current values of nodes without the drawing the nodes and edges.

8.7 Summary

In this chapter, we have discussed the research, development and use of DMT. DMT provides a means to visualise and manipulate dependency, locational and contextual structures that commonly exist in EM models. The main contributions of DMT are features to help model comprehension and reuse. In the case of model comprehension, we can trace the dependency of an observable easily by the feature of automatic dependency highlighting. We can import an existing definitive script and explore the dependency within the script interactively. In addition, we can use the concept of abstraction to represent contextual structure discovered in the model. In the case of model reuse, DMT provides interactive ways to extract and combine EM models based on well-defined strategies. We have also discussed the scalability issue of DMT and the limitations in visualising procedural and dynamic dependencies. We have also described some possible further researches and developments.