Chapter 5

Implementation to Support Distributed Empirical Modelling

DEM aims to enable all modellers in a distributed environment to interact collaboratively with each other in order to establish a working or shared understanding. Such understanding is crucial for improving interpersonal interaction in general and for shaping the agency of agents whose roles are associated by A-modellers in particular. In order to provide a distributed environment for collaboration, a tool supporting DEM (discussed in the previous chapter) is essential. This tool should be able to support the individual modelling task of each modeller, the interaction between modellers, and the interference of the S-modeller in this kind of interaction. It is clear that the tool tkeden[1], created to support individual modelling in EM, is unable to cope with all these needs. This motivates the creation of a new tool to satisfy the further requirements.

5.0 Overview

This chapter discusses the implementation issues involved in creating a tool to provide modellers with a collaborative distributed environment for supporting DEM. The tool

[1] Since the scripts of Donald and Scout are interpreted by Eden in tkeden and dtkeden, the term Eden will be used to represent the integration of Eden, Donald and Scout in this thesis, except where indicated.

tkeden, described earlier (Section 2.2), supports only the individual modelling task of the superagent within the framework of EM. Obviously, this functionality is not sufficient for a distributed environment supporting collaborative interaction between multiple modellers. On the other hand, the core part of tkeden, that is, its dependency maintainer, is still very useful for providing the environment with the advantages of definitive programming. In view of the limited time available for developing a new tool, the author decided to reuse this core part and extend it to a distributed environment.

A new tool called dtkeden has thus been developed by the author on the basis of the dependency maintainer implemented in tkeden. With this tool, modellers are empowered to interact with their own computer model, which corresponds to their own observed world, in an open-ended, situated manner. This kind of human-computer interaction not only facilitates individual understanding through modelling but also promotes other modellers' understanding through network communication. In this way, modellers can interact with each other via computer-mediated communication in order to support their collaborative interaction in a distributed environment.

In section 5.1, the main technical issues arising from the construction of the networking environment for dtkeden are discussed. First of all, a star-type logical network configuration for dtkeden is proposed. Then, the popular concept of client-server communication is introduced as the fundamental mechanism of network communication within this configuration. By integrating this mechanism with the socket program-to-program protocol and the TCP/IP communication protocol, a two-way network communication is devised for dtkeden. The networking feature can be used to shape the agency of the agents within an application by integrating a social process and a cognitive process, as described in the previous chapter. In addition, the problem of synchronisation arising from network communication between computer models is

identified, and a solution (designed by the author) that involves using a request-wait-reply mechanism with the aid of queued service numbers, is proposed to resolve this problem.

Section 5.2 focuses on the implementation of support for diverse interaction modes between modellers. The interaction between modellers is one of the main themes for DEM. Different kinds of interaction should be supported by different interaction modes for serving particular purposes. For example, an open, shared topic may suit a many-to-many interaction style, but negotiation-seeking in order to solve conflicts between group members is often invoked in the context of one-to-one interaction. Within the network configuration proposed in Section 5.1, four interaction modes have been implemented in dtkeden: broadcast, private, interference, and normal.

In the *broadcast* model, an open, shared environment for the interaction between modellers is provided. Any interaction triggered by a modeller is propagated to other modellers. In contrast to the broadcast model, the *private* model provides each A-modeller with an individual channel for one-to-one communication. The *interference* model then enables the S-modeller to interfere with the interaction between A-modellers in order to create a new context for modellers. The mode that is most characteristic of DEM is the *normal* model, which enables each A-modeller, acting as an agent, to interact with others by accessing authorised observables. This interaction mode exhibits the key principles of a framework for DEM in their full generality, and highlights the concepts of ethnomethodology and distributed cognition in particular. Within dtkeden, these four modes are used alternatively and can be changed at any moment in response to the context of modellers.

The main topic discussed in Section 5.3 is the reuse of software components. Over the past decade, the concept of software reuse has become widely promoted in the software community as a way of reducing development and maintenance costs and increasing productivity [Pau97, Pre97, Pri93, Som95]. The traditional approach to

software reuse usually seeks to develop software components that can be reused in their entirety without adaptation (so called 'black-box reuse'). In practice, the application of such black-box reuse in SSD remains a major challenge for software developers [PF87].

Section 5.3 offers an alternative to black-box reuse: adaptable reuse, which involves the reuse of software components by adaptation[2]. This mirrors the way in which a human being reuses experience by adapting it to similar situations in everyday life. This adaptable reuse hopefully loosens the boundary of reusable components and makes reuse more widely applicable. A particular kind of adaptable reuse can be achieved in EM through the concept of virtual agent implemented in dtkeden. First, the concept of virtual agent is proposed, and then the method of using it to facilitate adaptable reuse is discussed. A new kind of observable – the generic observable (GO) – is proposed to represent a set of definitions (that is, a definitive script) that can be reused through adaptation in this sense. The remainder of this section discusses the difference between using Abstract Data Types (ADTs) and GOs for developing reusable software components.

5.1 Network Communication in dtkeden

As explained in Section 4.2, network communication has an enabling role in supporting DEM. Within the framework for DEM, the S-modeller, as the external observer, and the A-modellers, as the internal observers, are empowered to perform empirical modelling by means of interaction with each other as well as interaction with their own computer models. In order to provide such a distributed, computerised interactive environment,

[2] The concept of adaptable reuse should not be confused with so called "white-box" reuse. See Section 5.3 for more details.

network communication[3], connecting together all the computer models of modellers, thus has the highest priority in the implementation of dtkeden.

This section illustrates the technical issues and problems encountered in implementing the distributed architecture of dtkeden. In the first subsection, this architecture is discussed on the basis of a star-type network configuration and client/server communication. The problems raised by asynchronous communication are identified and solved by the new synchronous communication mechanism implemented in dtkeden. The details are given in the second subsection.

5.1.1 A Distributed Architecture with Client/Server Communication

Since network communication is now ubiquitous in everyday life, the investigation of distributed systems has become an increasingly important trend in the software community. A distributed system normally consists of a set of software components located on different machines[4] and a network allowing these software components to communicate with each other to produce an integrated computing facility [CDK94, Hug97]. One of the fundamental issues in developing a distributed system concerns the architecture of network communication. From the perspective of software development, the architecture involves at least two parts: a logical network configuration and the techniques of network communication. The former determines the distribution of software components of the system, and the latter enables these components to communicate with each other via a physical network.

[3] In this thesis, "network communication" and "data communication" are interchangeable terms to denote an information technique whereby electronic data can be transferred from one hardware device to another through physical network facilities.

[4] In principle, it is not really necessary for all software components to run on different machines, but their communication with each other definitely must go through a communication network.

Referring to the framework for DEM shown in Figure 4-6, a star-type logical network configuration – probably the most common of all configurations [CCH80] – is exploited for network communication in dtkeden (see Figure 5-1). This star-type network configuration represents a logical interconnection between software components and is independent of the network communication topology, which illustrates the physical interconnection between the hardware components in which the software components reside [Hug97]. From the perspective of developing distributed systems, the logical configuration of software components is much more important than their physical configuration [CCH80].

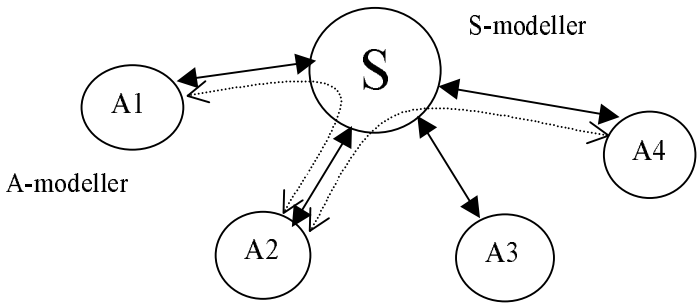

Figure 5-1. A star-type logical configuration for the network communication in dtkeden

There are two kinds of nodes in this star-type configuration. At the points of the star, A-nodes are created to represent A-modellers as shown in Figure 4-6. Each A-node can communicate with the S-node, at the centre of the star in Figure 5-1, via the network. Where the communication between two A-nodes is concerned, dtkeden does not provide a direct dialogue between them. Instead, this kind of communication is achieved through the involvement of the S-node, that is, by using the S-node as a message transferring centre to transmit the message[5] from one A-node to another. This kind of A-node-to-A-

[5] It should be noted that a *message* in dtkeden typically takes the form of a definitive script, possibly together with auxiliary functions and actions. Contrast the common use of 'message' in object-oriented programming to refer to a method invocation [Boo94].

node-via-the-S-node communication is provided through a built-in procedure in dtkeden. The modeller at the A-node can simply call this procedure to interact with the specified modeller at another A-node.

The S-node representing the S-modeller is responsible for the transmission of all messages between two A-nodes. This superior responsibility highlights the importance of the S-modeller as the only modeller (representing 'God's view') who is supposed to have direct access to all contexts. It also enables the S-modeller to intervene in the interaction between A-modellers. In this way, adopting a star-type network configuration has the advantage of enhancing the control of security and access privilege to observables, as explained in Section 5.2.

After determining the network configuration of dtkeden, the technique used to support network communication is taken into account. Following state-of-the-art techniques of network communication, the technique of the client-server (or request-response) model[6] is chosen to support the communication between computer-based models in the framework of DEM. The client/server model is currently the best known and most widely adopted system model for distributed systems [BG96, CDK94, KJ98, Som95]. It provides an effective general-purpose approach to the sharing of information and resources through network communication.

A typical client-server model, as shown in Figure 5-2, is oriented towards service provision. Within this model, each invoked client/server network communication consists of the following steps:

1. transmission of a request from a client to a server through the network;
2. execution of the request by the server;

[6] Although this term can also express a hardware-oriented view [BG96], it is used here to refer to a software-oriented technology.

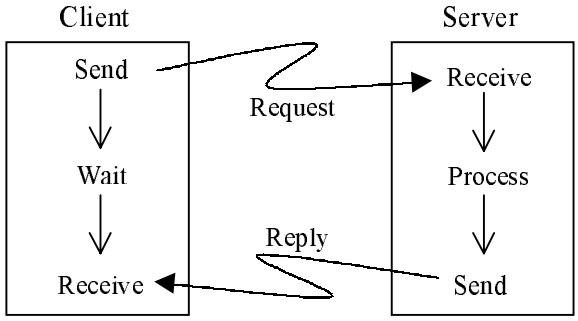

Figure 5-2. A typical client/server communication model

3. transmission of a reply to the client through the network.

This pattern of communication involves the transmission of two messages and a specific form of synchronisation[7] of the client and the server. As soon as a request is sent in step 1 to invoke a communication, the client is blocked until it receives a reply from the server in step 3. At the same time, the server must become aware of the request message sent in step 1 as soon as it arrives. After processing the received request, the server then sends its reply to the client in order to unblock the client and allow it to continue.

This elementary pattern of client/server communication can be implemented in terms of the message-passing operations *send* and *receive* as outlined above. Before sending a request, a client must know the names of the available servers and the services that they provide. Each request contains a communication identifier that is used to transmit the reply to the client. However, a server need not know either the identification of clients or how many clients there are. Typically, when a server starts up, it registers itself with a naming service, stating its network address and a global name for the service that it provides. A client obtains its network address by interrogating the naming service and is thus able to communicate with the server by using the same global service name.

[7] In this case, the synchronisation mechanism is called blocked communication. In other architectures, asynchronous communication may be involved: here the client is not waiting in suspense for the reply to be sent by the server.

Although client/server communication can be installed in various configurations - either centralised or highly distributed, as described in [BG96] - the communication of their components often follows this elementary pattern.

The implementation of dtkeden exploits the concept of protocols for program-to-program communication (or interprocess communication [Bac93]) that is widely used for implementing distributed systems [CDK94]. It enables direct dialogue between applications: for example, a C program calls a library function *send* to access the network communication layer. The library also offers functions for the initialisation of links, the connection establishment and breakdown, and the control of the data transmission itself, that relieve the programmers of most aspects of network communication. For example, the Socket protocol, used widely in distributed systems, provides such functions for a common communication interface. In dtkeden, the socket protocol provided by the Tcl/Tk[8] software package is used. Also, the TCP/IP protocol family, which is popularly applied to a local SUN workstation environment and available at Warwick, is used to provide the lower-layer network communication in dtkeden. An in-depth discussion of all these details is beyond the scope of this thesis.

In summary, the client/server communication in dtkeden is achieved by using the TCP network protocol and the Socket abstraction in the Tcl/Tk level. Figure 5-3 shows the layout of communication between the S-node and A-nodes, as implemented in dtkeden.

In some client/server architectural software systems, each client/server component is specified to be either a client or a server, but not both. Each component is only responsible for either requesting a service (that is, being a client) or providing a service

[8] Tcl is a scripting language and an interpreter for that language that is designed to be easily embedded into an application, and Tk is a graphical user interface toolkit for Tcl [Ous98, Wel97]. The tool tkeden uses the Tcl/Tk for the purposes of graphical interface control and visualisation, and dtkeden uses it also for dealing with network communication.

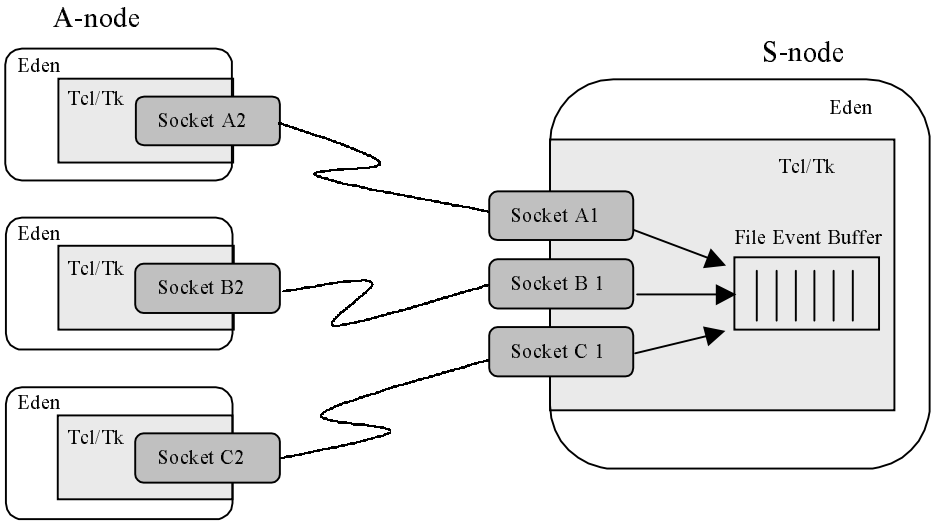

Figure 5-3 The communication between the S-node and the A-nodes

(that is, being a server). In such a case, there is only one-way, and not two-way, communication between components. For example, in the database world, the data repository and its associated functionality are termed the server; the client is the application (which could be on the same hardware or software, or not) [Aye95]. This exclusive attribution of client or server status indicates the one-way communication between the client and the server. In a similar fashion, for example, the Internet or WWW world has the web browser as the client, and the application providing the requested information as the server.

However, dtkeden refers to a client/server component in a different way. In dtkeden, a client means the requester of a service, and the server is the provider of a service. Any client/server component can be both a client and a server, depending on what it is doing at the time. In this case, 'client/server', as it applies to the implementation to support DEM, denotes a client/server communication technique rather than a client/server architectural configuration as described in [BG96]. Therefore, each node in Figure 5-1 is not exclusively specified to provide or request services. Instead, each node is devised to perform both functions in order to provide its user, that is, the modeller, with

two-way rather than one-way interactive communication. That is to say, each modeller in any node of either kind can communicate with other modellers by sending a request or providing a service.

An interesting comparison can be made between the client/server architecture of dtkeden and the information sharing architecture that is used in Empirical Worlds, recently developed by R. Cartwright for EM in [Car98]. In Empirical Worlds, information sharing is achieved by centralising all definitive scripts to the server machine so that clients can access the scripts via a 'definitions-database' system. The authority of each client accessing these definitions can be specified through a system of security control similar to that of the UNIX file system. In that case, clients can connect to the server in order to perform a modelling task. They do not need to communicate with each other, since each of them remains essentially independent. Any task for modelling, such as giving (re)definitions, is executed only in the server and displayed in the client. This centralised modelling environment allows each modeller to undertake exploration and experiments independently but to share commonly used definitive scripts in a collaborative manner. Obviously, such architecture, though having the advantages of sharing information and reusing definitions, provides limited support for DEM.

5.1.2 Synchronous Communication for dtkeden

One of the major challenges involved in the implementation of dtkeden arises from the need to tackle its synchronisation problem of client/server communication between nodes via a communication network. Before discussing this issue further, it will be helpful to re-examine the way in which intercommunication between Eden and Tcl/Tk is integrated with the execution of Eden.

As already explained, tkeden is a hybrid tool combining Eden and Tcl/Tk. Eden is an interpreter developed in C program language, and the Tcl/Tk is a popular tool for

event-driven programming. In tkeden, the tool Tcl/Tk is applied to deal with issues of external presentation such as the user interface, graphical display and window interaction, whilst Eden then focuses on handling the internal representation, for example, dependency maintenance and data manipulation. Since the Tcl/Tk is an event-driven tool, most of the time it is in the so called *EventLoop* loop, where the Tcl/Tk interpreter is devised to keep track of the condition of each event handler already issued in order to trigger its further actions when the condition is satisfied. For example, when Tcl/Tk finds that a button is pressed, the event handler *ButtonPressed* will be invoked to undertake specified actions. In the case of tkeden, the event handler *DoWhenIdle* has been specified to call a function of Eden, when Tcl/Tk is in an idle state. Hence, the intercommunication between Tcl/Tk and Eden is achieved by programming each to call functions of the other, so that program control passes forward and backward between them.

Two queues have been devised to hold received messages in Eden. One, called the executing queue (EQ), holds those messages to be executed. The other, called the waiting queue (WQ), stores the received message waiting to be moved into EQ. There is a function to process messages in EQ and then move the contents of WQ into EQ. It is this function that is triggered by the event handler *DoWhenIdle* when Tcl/Tk is in an idle state.

When tkeden is started, Eden builds up its initial state, then passes control to Tcl/Tk and awaits further actions invoked by the received messages. After taking program control from Eden, Tcl/Tk, as explained above, enters the EventLoop in which Tcl/Tk keeps track of the conditions of all issued event handlers in order to trigger further actions specified by these event handlers. Now, if Tcl/Tk receives a message, for example from the input window where the user can enter scripts, Tcl/Tk calls a function in Eden, and passes it the program control and the received message. This function then appends the received message to WQ and returns the control to Tcl/Tk. When Tcl/Tk enters an

idle state, the event handler *DoWhenIdle* calls a function in Eden and passes the program control to Eden. As mentioned above, this called function then executes each message holding in EQ and after the execution moves the messages in WQ into EQ. Finally, Eden returns control to Tcl/Tk and again Tcl/Tk enters the EventLoop. Figure 5-4 illustrates this intercommunication mechanism.

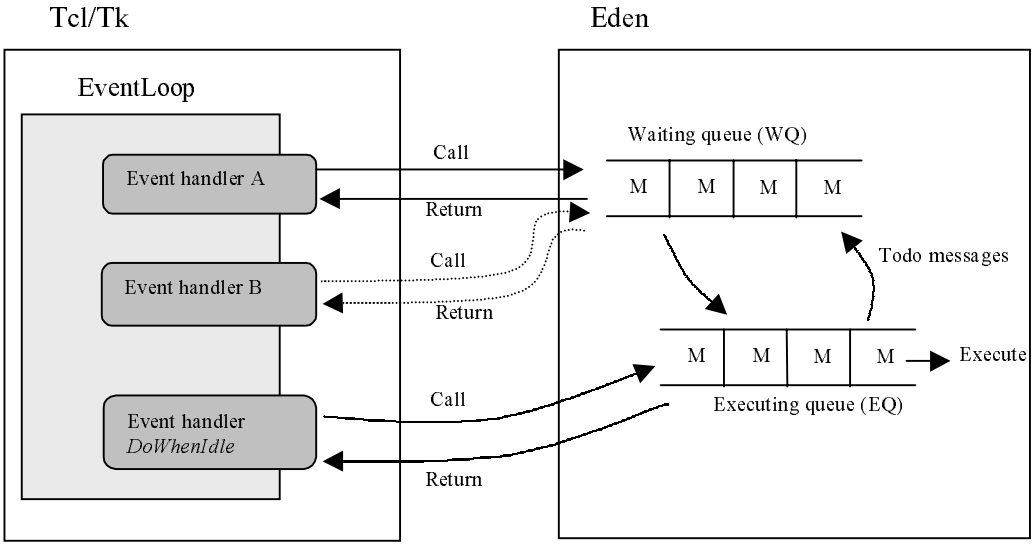

Figure 5-4. The intercommunication mechanism between Eden and Tcl/Tk

The main concern here is that Eden does not execute messages appended to its WQ until its EQ is exhausted. This feature allows Eden to deal with messages in a state-based fashion, that is, all messages arriving in the same conceptual state are executed at the same time. In particular, this mechanism underlies the development of a virtual machine for dependency maintenance, which is the most important and fundamental principle for designing a tool to support the framework of EM. Therefore, all auxiliary actions arising from dependency maintenance (e.g. as a result of triggering actions) are appended to EQ rather than WQ. In this way, it is as if they and all their preceding actions are executed in the same state (or conceptually at the same time). In practice, it is clear that genuine concurrent execution is not possible for Eden, but by this mechanism concurrency can be

achieved conceptually by separating messages arriving in one state from those that arrive in the next state.

In effect, as soon as Eden moves messages in WQ into EQ, it enters a new state. Until all these messages, including those moved in from WQ and those incurred from dependency maintenance, have been processed, Eden is viewed as being in the same state. This is the reason why two queues are needed. In this mechanism, it is evident that each time Tcl/Tk passes a message to Eden, it is appended to WQ rather than executed. In other words, the reply sent back to Tcl/Tk is not a confirmation that a message has been processed, but only an acknowledgement that a message has been received. This acknowledgement does not guarantee that processing of the received message will be immediately carried out.

In stand-alone use of tkeden, this mode of acknowledgement does not cause too many problems in terms of synchronisation. This is because Tcl/Tk still needs to wait for the execution of the acknowledged message after receiving the reply of acknowledgement. In tkeden, the only data stream from which input messages can be obtained is the interface between the modeller and Tcl/Tk, e.g. the input window devised for the modeller to input scripts. The messages received from this data stream can only be passed to, and processed by, Eden in sequential order. To be precise, these messages are handled by Tcl/Tk and Eden one at a time since tkeden is not a real concurrent system. This one-at-a-time mechanism forces Tcl/Tk to wait for the execution of the last message in Eden before it is available to receive another message. As a result, Tcl/Tk sometimes has to be suspended when Eden is dealing with a substantial computational task (such as a complex evaluation). Despite this, the synchronisation of Tcl/Tk and Eden in the same tkeden can still be achieved. However, for dtkeden, the situation is different. The mechanism of receiving and processing messages between Tcl/Tk and Eden does not

work in the same way as in tkeden, because more than one machine with similar interfaces between Tcl/Tk and Eden are involved.

First, in addition to the original window-based interface, another data stream is provided by dtkeden for collecting messages. This stems from the sockets implemented for client/server communication. In dtkeden, the socket technique for program-to-program protocol is used to establish communication channels in the Tcl/Tk level to receive/send messages from/to other machines via the communication network. Hence, within a machine running dtkeden, the collected messages can come from other machines. In other words, messages requesting services can come from both kinds of data streams, and can be mixed up together and executed by Eden.

An event handler *fileevent* is implemented to deal with the tasks of reading/writing a message from/to a socket following this event handling strategy mentioned above. For example, when a message requesting a service arrives at a socket in the server, a signal is sent to Tcl/Tk to start the event handler *fileevent* of this socket. This event handler then buffers the received message and passes it to Eden. As soon as the message is received, Eden appends it to the WQ and passes program control to Tcl/Tk with an acknowledgement, as described above. This acknowledgement then leads *fileevent* to reply to the client with a confirmation.

This message-passing mechanism for remote communication (as illustrated in Figure 5-5) is essentially asynchronous, though it still takes the form of a kind of synchronous message passing. It does not guarantee that Eden in the server has accomplished the requested service when the client receives the confirmation reply. In effect, it just acknowledges that the request has been appended to WQ in Eden. This mechanism is not the same as typical synchronous client/server communication in which the client waits for the server's reply in order to confirm that the requested service has been executed.

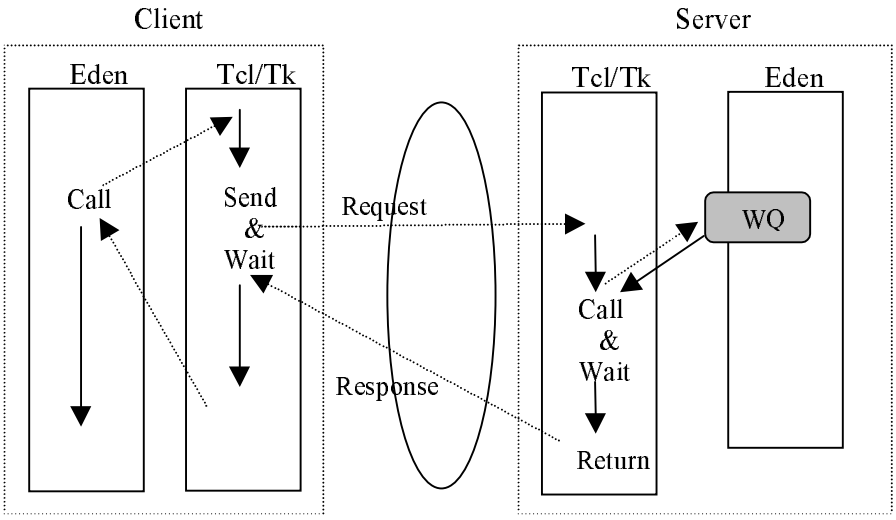

Figure 5-5. The asynchronous communication in dtkeden

An even worse situation can arise in a distributed environment, when two or more clients are sending various requests to the same server. Figure 5-3 shows a situation in which many senders are connected to one receiver. In this case, there are as many sockets in the server as there are senders in order to distinguish individual connections between each client and the server. This disambiguation of messages does not resolve all communication problems, however. Time delay is unavoidable and the sequence of executing the received requests is unpredictable. After all, the Tcl/Tk interface to the server can only handle one event at a time. It should be noted that an event handler *fileevent* is created for each socket in the server and in the client.

Therefore, problems of synchronisation between the sender and receiver arise, especially when modelling distributed, real-time systems in which timestamps in connected machines have to be consistent. For example, in a case study of modelling a railway accident (discussed in the next chapter), the same train's position has to be displayed at the same place on the screens in different machines running dtkeden. The asynchronous communication between these distributed dtkeden environments potentially leads to some unacceptable situations: for example, the trains may be

displayed at different positions; some of them may stop and then jump to a new position instead of moving smoothly, and so on.

In fact, such problems of synchronisation could be avoided if the client/server communication through sockets were to be implemented in the Eden level rather than in the Tcl/Tk level. In that case, the client and the server will be totally suspended until the requested service is provided, since there is only one thread in the server to deal with the requests of clients. That is to say, no more requests can be accepted before the current request has been served. This limitation on processing of requests leads to inconsistency in the suspended components' states and other components' states. To resolve this problem of inconsistency, multi-thread or event-handling techniques are needed for dtkeden. However, implementing these new techniques using the C language could involve more unpredictable challenges, such as problems of concurrent control. Due to the limitations of research time, the decision to implement client/server communication in the Tcl/Tk level was made. This makes the problems of asynchronous communication identified here inevitable, and a solution is therefore needed.

To overcome the problems posed by asynchronous communication in dtkeden, a request-wait-reply (RWR) mechanism was devised to improve the existing message passing mechanism provided by Tcl/Tk and Eden. This new synchronous mechanism for remote communication in dtkeden, illustrated in Figure 5-6, will not allow the server to reply immediately to the client with an acknowledge message when it receives a request that needs to be served synchronously. Instead, the event handler *fileevent* in the server will enter a state to wait for the accomplishment of the requested service, even if it is queued in WQ in Eden. This waiting situation in the server will consequently lead the event handler *fileevent* in the client to enter a waiting state as well, since the communication between these two fileevent handlers via sockets is essentially synchronous.

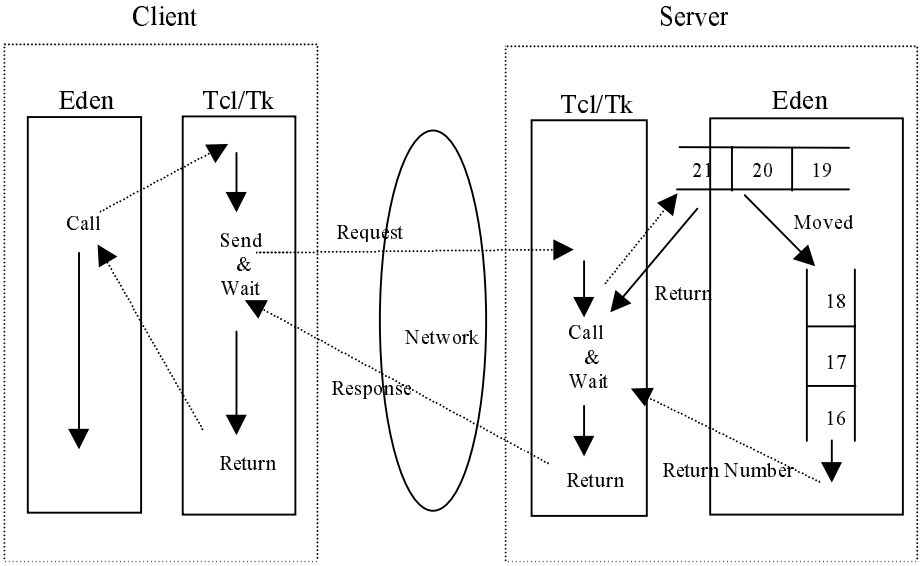

Figure 5-6. A synchronous model for remote communication in dtkeden

It should be noted that when they are in a waiting situation, neither the server nor the client is completely suspended. Only partial components, that is, the pair of sockets connecting the communication between the server and the client, are suspended: for example, the pair of sockets socketA1 and socketA2 shown in Figure 5-3. The partial suspension enables the client and the server to continue to deal with messages from their other input streams, such as the graphical interfaces and sockets in the Tcl/Tk level. Only when the requested service is provided does the server send the client a confirmed reply to release both suspensions. The need to minimise the suspension of the client and the server in their synchronous communication is the main reason why the communication sockets are implemented in the Tcl/Tk level rather than in the Eden level.

One of the crucial issues to be addressed in implementing this RWR mechanism is how the event handler *fileevent* knows that its request has been served after entering a waiting state. Within a single client/server communication, as illustrated in Figure 5-2, the server can only serve one request at a time. When it is busy serving, other requests must wait in a queue for their turn to be executed. Once a request is executed, the server

continues to process the request without delay and interruption until the request is completed. Hence, when a request is accepted, the client sending this request clearly knows that the server is processing the request, and the server also knows to whom the reply should be sent.

However, on the account of dependency maintainer in tkeden, the processing of messages in dtkeden is state-based rather than sequential. As explained earlier, each request arriving at the server must be queued in WQ rather than be executed immediately. Because of this accepted-first-served-later feature in Eden, the event handler *fileevent* loses contact with its request. If no account is taken of this fact, RWR mechanism devised above can keep the *fileevent* waiting for a reply, but the *fileevent* cannot recognise whether or not its request has been served, since the linkage between the *fileevent* and its request has disappeared.

The method of using a sequence of numbers to arrange the order of service (a popular method in everyday life), is applied to deal with this problem. Each time *fileevent* event handler passes a request to Eden, Tcl/Tk issues a service number to the event handler and suspends it for further reply. At the same time, Tcl/Tk passes the request plus the service number to Eden. When Eden processes a numbered request, it replies to Tcl/Tk with the service number. Once the replied number is equal to the issued number of an event handler, Tcl/Tk releases the suspended *fileevent* immediately. The fileevent in the server is then available to reply to the client with a confirmation. After receiving this confirmation, the fileevent in the client is also released. In other words, a synchronous communication between the client and the server is achieved. The number on each message square in WQ and EQ (shown in Figure 5-6) serves to illustrate how this concept of queued service number operates.

5.2 Interaction Modes in dtkeden

As explained in earlier chapters, the human factor is arguably a critical issue within software system development (SSD) [LR98, Som95, Flo87]. Interpersonal interaction amongst participants is one of the main resources for guiding and shaping the process of SSD and is therefore a key factor in determining the success or failure of a SSD project. It is widely recognised that inefficient and ineffectual interaction between participants is one of the major sources of confusion and error in SSD [STM95].

In dtkeden, interaction between modellers is achieved through networked computer models whose architecture and mechanism for network communication have been discussed in the previous section. Within such a computer-based distributed modelling environment, the interaction between modellers is computer-mediated and may involve no face-to-face interaction. Within the computer-mediated interaction, modellers cannot necessarily look at each other or use verbal or body language for interaction. They may not know each other. Instead, their networked computer models become the communication medium for the interaction between modellers. The visualisations of these models represent the construals of the modeller. In such interaction, each modeller 'speaks' through changing their computer model. Such a change is passed through the communication network and affects the computer models of others (the 'listeners'), so as to 'tell' them what the speaker is thinking.

Computer-mediated interaction is not subject to the same temporal and spatial constraints imposed by face-to-face interaction, but is recognised to be less effective and less socially rewarding [GK94]. On the other hand, compared with paper-based interaction, computer-mediated interaction provides modellers with active assistance in searching, understanding and creating knowledge in the course of co-operative problem

solving processes [Fis91, DS97]. Indeed, it is hard to say which mode of interpersonal interaction is best suited for modellers, since real-world situations vary considerably.

Although in technical respects the architecture of dtkeden is based on the framework of DEM, both E-modelling (which is concerned with the interaction between modellers acting as external observers) and I-modelling (which is concerned with the interaction between agents enacted by A-modellers as internal observers) are supported by the current version of dtkeden. Both kinds of modelling require modellers to interact with each other for shaping agency and exploring the mutual understanding between modellers in a distributed environment. As pointed out by D. Sonnenwald [Son93, Son96], diverse interactions between group members in the design process are required in order to develop a comprehensive understanding of design and facilitate multiple exploration of knowledge. Sonnenwald identified a variety of roles and interaction networks for intergroup and intragroup members in each phase during the design process in order to highlight the diversity of interaction styles between all modellers. In the context of a large system, the architecture of the interaction among all modellers from multiple disciplines, domains and individuals can become exceedingly intricate [Son96]. One possible way to support such interaction is to decompose it into a number of small group interactions. Each small group interaction could possibly be supported by a distributed computational environment such as dtkeden. Figure 5-7 illustrates how such decomposition could be based on the framework proposed for DEM[9]. It should be noted that the intergroup communication between intergroup stars (in D. Sonnenwald's terms in [Son93]) is not yet explicitly supported in dtkeden.

[9] This illustration is based on D. Sonnenwald's work on intergroup communication in the planning phase among intergroup stars in the user, designer and developer group [Son93, p. 63].

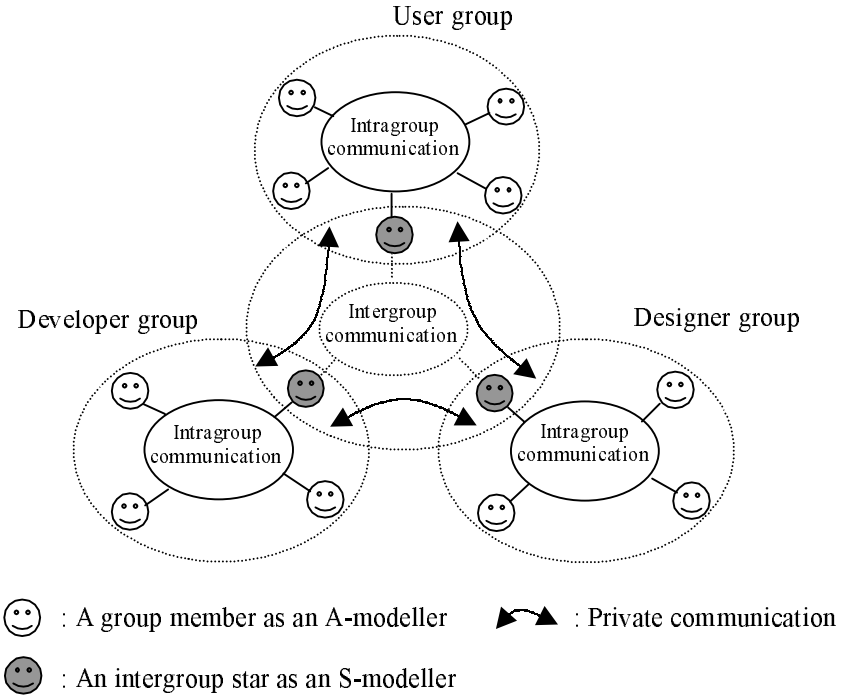

Figure 5-7. Decomposing large group communication into small group communication on the basis of the DEM framework

To support the interaction between modellers in forms of I-modelling and E-modelling in a distributed environment, four interaction modes for the S-node[10] have been implemented in dtkeden: broadcast, private, interference and normal.

The *broadcast* mode is the most primitive style of interaction between modellers. Its broadcasting mechanism is established by means of the S-node performing the role of a message-transferring centre. According to the star-type logical network configuration that is implemented in dtkeden, each message sent from an A-node must first come to the S-node. If the S-node at that moment is in the broadcast mode, the arriving message will be automatically broadcast to all other A-nodes, and this will consequently change the visualisations of the computer models at these A-nodes. The modellers at these A-

[10] In the current version of dtkeden, the interaction mode between A-modellers is established only in the S-node since it is a message transferring centre (see Section 5.1). Each A-node only sends and/or receives messages, but does not transmit any message. In addition, an A-modeller can change the interaction mode, if the necessary privilege is given.

nodes then construe this change and take this into account in their own computer models when invoking further interaction.

In as much as modellers co-operating in the broadcast mode are explicitly informed of changes, they can be regarded as being in an electronic group meeting without video and audio support. They share all messages sent out from any modeller by means of propagation, and typically interact with each other in an iterative manner in order to reach consensus. Interaction of this nature is common in the interaction between group members, such as in most inter- and intragroup interactions during the design process proposed in [Son93]. This mode can be used for developing a system, such as a multi-user game, that requires a shared environment for supporting the interaction between its users. (Examples are given in Section 6.3.)

The *private* mode provided by dtkeden supports a one-to-one interaction between the S-node and an A-node. As D. Sonnenwald observed from empirical studies, in many cases, one-to-one interaction plays an important role for managing different perspectives in a group [Son93]. To support this one-to-one interaction, the private mode provides each modeller at A-nodes with a private interaction channel to the S-modeller at the S-node. In the private interaction mode, in contrast to the broadcast mode, no message will be propagated to the other A-nodes. Since it is possible for more than one such private channel to exist in parallel, this private mode is also suitable for many-to-one modelling environments, such as is required in a system monitoring student learning (see Section 6.3). It should be noted that the interaction between two A-nodes is achieved through a built-in procedure providing A-node-to-A-node-via-the-S-node communication, as discussed in the previous section (Section 5.1).

The *interference* mode is a very useful mode for modelling a situation through 'what if' experiments. It allows the modeller at the S-node, acting as the superagent, to interfere directly in the interaction between modellers. Before being serviced, each

message arriving at the S-node is displayed on the input window of the modeller at the S-node. The superagent can exercise discretion over how the suspended message is processed by changing its content. The 'what if' experiment enables the superagent to explore and experience an unfamiliar context for the interaction between modellers. This often leads to unexpected situations that can provide modellers with surprising and enriching discoveries.

Interference mode is of particular interest for modelling the phenomena of the real world, where many-to-many interactions are the norm and singular conditions require the intervention of a God-like superagent (cf. [Bey98]). It is also applicable to the case in which the interaction between modellers necessitates multi-faceted exploration in order to improve comprehension among modellers at the S-node and A-nodes. For instance, the need for such mutual exploration has been identified by Sonnenwald in connection with the creation of innovative artefacts [Son96]. In addition, high-level managerial interference in the interaction between lower-level personnel in an organisation is also a suitable application for the interference mode.

The default interaction mode in dtkeden is called *normal* mode. In this mode, the interaction between modellers is mediated by the computer with reference to specified privileges of modellers to access observables. For E-modelling, the access privileges of modellers typically are given so as to reflect management control and social relationships between modellers. For example, in a design process, users are often not allowed to change those observables associated with implementation, and designers from different groups can be restricted to access different observables (cf. [ABCY94]).

Normal mode is particularly significant for I-modelling, where modellers are required to interact with each other in the roles of agents. As explained in Section 4.2, their interaction, which is invoked in the form of pretend play, should reflect the agency

of the individual agents whose roles they are playing, as defined by their privileges to change observables.

A privilege to act is specified by a guarded action such as is used in the *protocol* part of an LSD account. The generic form for such a guarded action is as follows:

guard → action

This guarded-action can be interpreted as asserting that the agent has context-dependent privilege to undertake the state-changing *action* if the *guard* is true. A privilege to act implies that certain privileges to access those observables that are involved in the action are also needed. That is to say, in order to exercise its agency, the agent must be able to gain access to the observables involved in the *guard* part for observation and to those in the *action* part for modification. The agency of an agent is suspended if the agent does not have the appropriate access privileges.

Access privileges are significant for the interaction between agents. They provide a richer model of context-dependent agency than guards alone supply. As explained and illustrated in Section 4.3, agency can appear and disappear as the context surrounding an agent changes. Taking the context of agency into account helps to make the computer model resemble the referent in the real world more closely, as is needed in order to clarify the modellers' understanding.

In previous work on EM, the access privileges to observables by an agent have been described in the *oracle* and the *handle* parts of an LSD account respectively. The design of tkeden does not take this description into account. This is partly because the core part of tkeden is essentially observable-based. The agent concept is only partially represented by triggered action in tkeden.

A more important reason is that in such a stand-alone modelling environment the unique modeller of EM, being a superagent, is empowered to access all observables.

However, when tkeden is exploited in a distributed environment, the agents described in an LSD account are distributed on different workstations. Any observable could be accessed by the superagent S-modeller but also by A-modellers acting as agents via a communication network. In this case, the access privilege to an observable by an agent should be considered carefully. After all, in the real world, observables are often not open to all agents. For example, in a two-player draughts game developed in dtkeden, it is not appropriate for a player to be able to change the positions of his/her opponent's pieces except in a situation in which the player has captured the opponent's piece.

To deal with access privileges to observables by agents, a system agent called *LSDagent* has been implemented in dtkeden by the author. The current version of dtkeden can take two kinds of access privilege to observables into consideration: *oracle*, to specify which observables an agent can access for observation; and *handle*, to specify which observables an agent can conditionally change. When an agent is attempting to access an observable for observation or modification, the LSDagent checks whether or not this agent has the necessary permission to access this observable. Without valid permission, an agent is not allowed to observe and/or modify an observable.

The author has extended dtkeden to enable the S-modeller to specify agent's access privileges to observables that are described in the oracle and the handle parts of the LSD account [Bey86]. This extension involves introducing an LSD-based notation into dtkeden to provide for the declaration and cancellation of access privileges. The details of using the notation are given in Appendix 5-A. With this LSD-based notation, the S-modeller is empowered to declare or remove the privileges of agents to access observables with reference to the LSD account. Accordingly, agents' access privilege to observables can be established in the computer model of the S-modeller. With the specified access privilege, the LSDagent in the computer model of S-modeller can verify the permission of an agent to access observables, and can propagate the change of an

observable that is changed by an agent to other agents who have access privilege to the observable for observation.

In the real world, an agent's access privilege to an observable is not always persistent but can be mobile and volatile. Considering the draught game example above, when a player's piece jumps over the opponent's piece, the player can remove this piece from the board. In this case, the player gains temporary control over the position of his/her opponent's piece. Similarly, after this piece has been removed from the broad, there is no significant sense in which either player can change its position again. In other words, an agent's access privilege to observables changes dynamically. Functions for the purpose of changing the access privilege on-the-fly have been implemented in dtkeden. Their details are also given in Appendix 5-A.

It should be noted that the implementation for supporting normal mode is different from the implementation for the other three interaction modes in dtkeden. The normal mode is implemented in the Eden level, since agents' access privileges to observables are associated with the internal representations of each observable and each agent, and their interrelationships. The other modes all concern the mechanism of message passing (that is, the protocol for the transmission of a definitive script) and can be implemented in the Tcl/Tk level.

With the star-type architecture that is implemented in dtkeden, an interesting issue emerges when multiple modellers are involved in a distributed environment: which modeller should be at the S-node? As explained earlier (Section 5.1), the role of the modeller at the S-node is to be the superagent transferring messages that occur in the interaction between modellers at A-nodes and potentially interfering with these messages. Thus, it is appropriate for the modeller at the S-node to play a more powerful role, such as that of the manager, the intergroup star, or intragroup star, in order to highlight the modeller's characteristics.

In addition, the interaction between modellers is not always in one of the above, specific interaction modes. It can instead be switched dynamically between these modes by programming or choosing different menu items in the input window at the S-node. In practice, the interaction between modellers can be more subtle than these four interaction modes can provide, even in combination. Further improvement is suggested in Section 8.3 for future work.

5.3 Adaptable Reuse in dtkeden

Over the past decade, the reusability of software components has become widely publicised in SSD, since it has been proved to be helpful in reducing development and maintenance costs and increasing productivity [Som95, Pre97, Pau97]. The traditional approach to software reuse usually seeks to develop software components for complete reuse without modification. This demands that the reused component be completely fitted into a new solution domain. In practice, it is very hard to find two solution domains that are exactly the same [PF87]. This difficulty leads to a major challenge of putting reuse into practice and integrating it into software development processes.

As described earlier (Section 2.2), each definition of an observable in the computer-based model is the modeller's construal of the observed world. In the context of the A-modeller, the definition is unique. However, the uniqueness may become problematic when it is sent to the computer-based model of the S-modeller. This is because within this model there may exist more than one definition for the same observable. These definitions indicate that the same observable can be construed in different ways by different A-modellers in a distributed environment. For example, an A-modeller X may define an observable M as 'M is A+B', but another A-modeller Y may define the same observable M as 'M is A+C'. Obviously, a problem of inconsistency between two definitions of the observable M arises, when both are sent to the S-modeller.

In a conventional computational framework – and indeed in the stand-alone environment of tkeden – such inconsistency is not allowed, since a variable cannot have two different definitions (or internal representations) at the same time. A mechanism by which a later definition overwrites an earlier one may be invoked. In some context, further interaction between X and Y or an appeal to arbitration will be invoked to eliminate the inconsistency.

However, in the real world, the elimination of inconsistency is not absolutely necessary. The coexistence of different definitions is sometimes needed, for example, for the purpose of distinguishing individual differences in perception. Taking the same example above, to clarify the difference between the two definitions given by A-modellers X and Y, the S-modeller may want to display the observable M's content by evaluating the two definitions simultaneously in various situations. In this case, the need to keep both local definitions co-existing in the model of the S-modeller becomes clear. In practice, even in a local model, it is sometimes necessary to separate others' definitions from the modeller's own for the same observable.

Moreover, from the perspective of distribution, it is better to prevent the contexts of all modellers from becoming mixed up together in order to keep track of the individual context. To achieve this, all definitions given by modellers must independently co-exist, otherwise the overwriting mechanism will be invoked to eliminate the inconsistency between different definitions of the same observable. However, the internal representation of a variable in a program can only be described by a definition at one particular moment, so that it is not possible to give an observable multiple definitions that co-exist at the same time. The notion of *virtual agent*[11], which provides a means to

[11] For convenience, the virtual agent is usually given the name of the A-modeller providing these definitions, though it is not necessary to do so.

associate a family of definitions with an agent, is introduced into dtkeden in order to cater for the demand for co-existent definitions.

Conceptually, virtual agency provides a way of attaching a definitive script to a particular observer, typically so as to represent the personal perceptions of that observer. Ideally, the association between a script and its observer should be defined and manipulated as a form of dependency. For instance, it would sometimes be convenient for one agent to hand over a script to another. In practice, there are serious technical difficulties in implementing such a feature in tkeden. This means that virtual agency is managed in dtkeden in a procedural fashion.

Definitions in a dtkeden script are associated with a virtual agent according to the context in which they are introduced. If no virtual agent context has been declared, definitions are in the *root context*. As soon as a virtual agent is declared, dtkeden shifts its current context to the context of this virtual agent and then localises each subsequent definition until another new context is required. The *localisation* of a given definition means that each observable used in the definition will be renamed by appending the virtual agent's name to its original name. The renamed definitions then can be distinguished from those given by others. For example, with this virtual agent notion, if virtual agents are given the names of their respective A-modellers, the two above definitions for the observable M can be localised as 'X_M is X_A + X_B' and 'Y_M is Y_A + Y_C'. Both these definitions are present in the computer model of the S-modeller.

The virtual agent concept is helpful for creating a distributed model. For the S-modeller, the different definitions of the same observable associated with different contexts can co-exist in the same model, and can be easily accessed by declaring the context of a virtual agent. A definition made by the A-modeller X in his/her local context is transmitted to the S-node and, through localisation, is interpreted as if it were introduced at the S-node in the context of a virtual agent X. This mechanism allows each

A-modeller to use observables in his/her computer model without needing to take special steps to guard against ambiguity. An A-modeller's local context is independent of the localised context in the S-modeller's model, because the localisation mechanism is invoked after the message is passed to the S-modeller and before it is internally represented in the S-modeller's model. Appendix 5-B illustrates the syntax of using a virtual agent to shift context in dtkeden.

In effect, given a virtual agent and a set of definitions (that is, a definitive script), the localisation mechanism can be regarded as a mechanism for generating a new definitive script associated with the context of a virtual agent. The generated definitive script with renamed observables is different from the original one so that dtkeden will store all these new definitions and maintain their dependency automatically. It should be noted that there is no defined dependency between both definitive scripts after localisation.

Hence, with the virtual agent notion, a definitive script can be used as a pattern to generate different definitive scripts associated with different contexts. This conclusion is very valuable for dtkeden when the reusability of a definitive script is taken into account.

In reusing a definitive script, an ontological problem concerning observables in EM and DEM emerges. Following [Bey98], it appears that each observable in a definitive script must correspond to a characteristic of the modeller's external environment [Bey98, SB98]. Taken at face value, this means that each observable must be conceptually subject to an object in the environment, including the observed world and the computer model; otherwise the modeller cannot observe it[12]. According to this rule, for example, the corner of this table and the status of my bank account, are observable, but the corner of *a* table

[12] In fact, [Bey98] involves an extended discussion of how the notion of an observable in EM embraces entities that are quite different in character from the physical observables of commonsense. To observe an observable in EM and DEM need not mean to physically "see" this observable. Instead, it can be used in a broader sense to refer to entities that are typically construed, for example, as imaginary.

and the status of *a* bank account are not observables because no particular object is identified for the modeller's observation.

By this interpretation, an observable in EM can only be described in the details of the modeller's observation. It is quite inappropriate to reuse an observable of this kind because it has a specific referent. This is the reason why most systems developed by EM contain a huge numbers of definitions. For example, Figure 5-8(c) illustrates several observables, called '1st door', '2nd door', ..., and 'nth door'. Each consists of a family of observables with similar characteristics. The similarities between them provide no help for reuse due to their specific context. In the same manner, in fact, no description of observables corresponding to the observed world can be reused as a pattern.

For the purpose of reuse, 'observables' that are not subject to a particular context corresponding to the modeller's environment are needed (cf. [GYCBC96]). A possible solution to this crucial problem of reuse in dtkeden is to use abstraction, which disassociates the significant characteristics of an object from any specific instance [Ber94]. Although abstraction can separate observables from their detailed context to serve the purpose of reuse, this separation has other implications that impose inevitable limitations on the scope and nature of reuse. Further details will be given later when adaptable reuse in dtkeden is compared with complete reuse based on an abstract data type.

A more appropriate solution, devised and implemented by the author in dtkeden, is to create a new kind of observable, called a *generic observable* (GO), for the purpose of reuse (cf. footnote 11). Unlike those observables that correspond to real world objects in the modeller's external environment (as described in Section 2.2), GOs are created to correspond to the modeller's experience, which is inside the modeller's mind and emerges from repeated description of certain observables with the same characteristics. For example, after repeatedly describing a number of doors with the same characteristics,

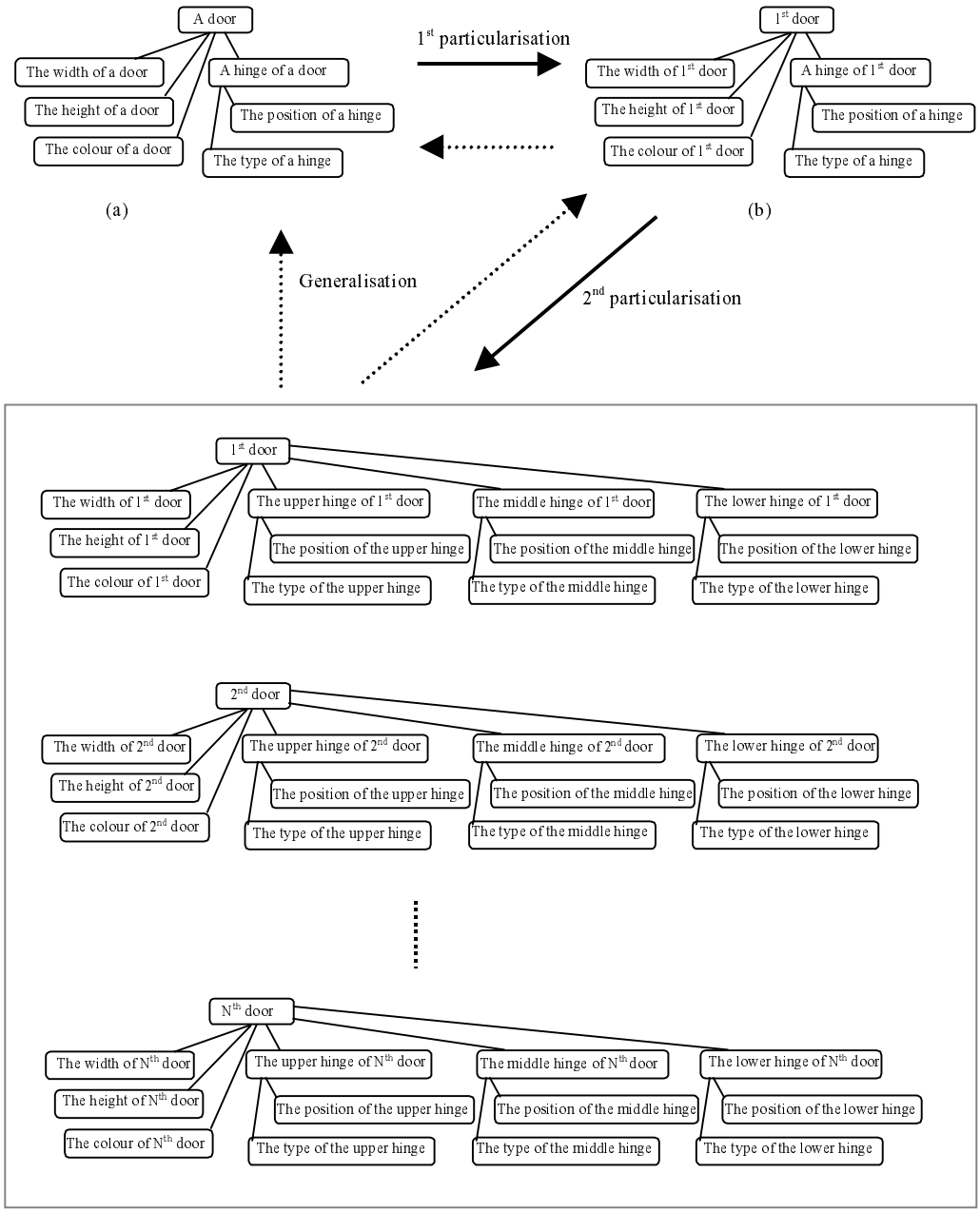

Figure 5-8 An example of particularisation and generalisation

the modeller can generalise a GO and give its description as illustrated in Figure 5-8(a) or (b).

The *generalisation* process, as shown in Figure 5-8, is a process in which similar descriptions are repeatedly given for certain observables with the same characteristics, so

that the modeller is able to create GOs and give their descriptions in response to the emerging experience. In this sense, the created GOs correspond to the modeller's practical experience rather than to particular objects in the modeller's external environment. Since the experience is obtained from the repeated description in practice, it is inappropriate to view this generalisation process as a form of abstraction, which is completely independent of practice. The key point is that the experience emerging from the generalisation process is shaped by the character of the modeller, as determined by his/her knowledge, intelligence and past experience. This is in contrast with the concept of abstraction in which personal characteristics are not taken into explicit account.

Since GOs are derived from the modeller's practical experience in a generalisation process, no guarantee can be given that they will emerge and be of adequate quality. Instead, they are contingent, volatile and unpredictable. The modeller, observables themselves and the generalisation process are all factors eligible to affect the creation of GOs. For example, a novice modeller may take a long time to form practical experience underlying the generalisation of GOs, but an experienced modeller may not. Also, the qualities and the content of the GOs' descriptions as generalised by both kinds of modellers could be very different. Figures 5-8(a) and (b) shows that different generalisation processes, from (c) to (a) or from (c) to (a) via (b), are produced, and this leads to the creation of different GOs, such as 'a door' and 'a hinge of my door'.

In this sense, the term 'reuse' in dtkeden in effect refers to the reuse of practical experience. The reuse of experience is one of the fundamental ways in which human beings cope with problems in everyday life. There are many examples: the driver finding his way round a city he has not visited before, the teacher teaching a new class, and so on. In such cases experience provides human beings with a great deal of help in solving situated problems.

Therefore, it is clear that the experience-oriented reuse of a definitive script is informed by the essential characteristics of empirical modelling, such as subjectivity and situatedness. This means that on the one hand, reuse in dtkeden can become uncertain, risky and not persistent. On the other hand, these characteristics do offer benefits, such as usability and adaptability, as suggested by P. Ness in [Nes97].

After creating a GO, its description, that is, a definitive script, can be used as a reusable pattern by combining it with a virtual agent. A new definitive script can be generated by localising the pattern. In order to use this definitive pattern over and over again, it is convenient to store it in a file. Each time the modeller wants to reuse this pattern, he/she just needs to include this file in his/her computer model and specify a virtual agent for localisation.

So far, only the mechanism by which a virtual agent can be explicitly specified from the input window in dtkeden has been described (see Appendix 5-B for details). Although the way in which a virtual agent is specified does not influence the semantics of a definitive script in reuse, a way to specify a virtual agent without the explicit involvement of the modeller is also helpful. This is because the context of the virtual agent may have to be determined and established in an automatic fashion. For this purpose, the name of a virtual agent can be specified implicitly by a string variable. With this feature, the modeller can store a definitive pattern in a file with an anonymous virtual agent as its header. In this way, the context of a GO can be dynamically determined.

With these mechanisms in place, the reuse of a definitive pattern in dtkeden becomes more flexible and applicable. As soon as an observable similar to the characteristics of a created GO is observed, the definitive pattern of the GO can be exploited for reuse. To reuse the definitive pattern, the floating context of the GO has to be particularised to the context of the observable. This *particularisation* can be achieved by three steps: specifying the undetermined virtual agent (identification), retrieving the

content of a GO (instantiation), and finally localising the retrieved content to the new context of this observable (localisation).

The *identification* step can be accomplished by the modeller by declaring a virtual agent interactively via the dtkeden input window, or by invoking a suitable procedure or function to change the virtual agent context. The *instantiation* step creates an instance of a GO by introducing a definitive pattern into the context established in dtkeden in this way. Following this instantiation, *localisation* of the definitive pattern is the process that creates an appropriate syntactic variant of the definitive script in the computer model. In this way, the definitive pattern of the GO is reused.

Experience is adaptable. Certainly, experience is not only reused for the same situation, but also for similar situations. For a driver, no driving situation is exactly the same as any previous driving experience. Too many factors, such as different roads, the presence of different cars and drivers, different traffic system, etc. add new dimensions to the driving situation. However, a driver still can drive in such different situations and does not need to relearn how to drive. This is because human beings can adapt past experience to a new situation, and indeed all past experience was originally derived from previous different but similar situations. Reusing experience is apparently more significant in everyday life than reusing well-defined but unchangeable program codes or even abstract components, such as specification and design [Som95].

Traditionally, there are two ways to reuse software components: black-box reuse and white-box reuse. The former refers to the reuse of well-defined components without modification, but the latter demands further change to the reused components. In practice, black-box reuse is of limited use, because it is very expensive to develop reusable components that are suited to a variety of situations without change [PF87, Pri93]. So far this kind of reuse is only applied in developing relatively low-level components, such as procedures for supporting the GUI (graphical user interface) and database framework.

Regarding white-box reuse, the difficulties mainly stem from the extra efforts for the necessary modification. This is because changing components can be formally thought of as engaging in parts of, or even the whole of, a software development process. The effort needed for this engagement can to a large extent offset the benefit of reuse [Pau97]. In addition, the trend in white-box reuse is towards parameterisation and built-in adaptability [Pri93]. This obviously makes white-box reuse more flexible and applicable, but it requires further formalisation of the intended components prior to reuse. The prior formalisation usually must be invoked before run time and accomplished in a context-free manner. As a result, problems similar to those of using the concept of abstraction for reuse emerge. For example, the extent to which inheritance, devised for white-box reuse in object-oriented programming, supports software reuse is subject to controversy [GHJV95, Som95].

By contrast, the reuse of definition patterns by adaptation is encouraged in dtkeden. In comparison with white-box reuse, there is less difficulty in changing the reused components (that is, the definitive patterns) in adaptable reuse, as proposed here. In the light of definitive programming, as described in Section 2.4.2, adapting the particularised definitive scripts is far simpler than the software development process in conventional programming. If any part of the reused definitive pattern is not well suited to the new context, the modeller can simply give new definitions to replace the unsuitable definitions. As described above, experience is adapted to a new situation in everyday life in much the same manner. For example, in reusing the definitive pattern of a GO called 'an action button' to produce a new action button, the caption of the latter has to differ from that of the original one. After localisation, by means of definitive programming, the modeller can easily change the localised definitions of observables associated with the caption simply by introducing new definitions.

In principle, stored definitive patterns can be reused over and over again to produce different definitive scripts. They can also be reused for other systems when the modeller faces a similar situation. This is one of the main reasons why experienced modellers usually spend less time accomplishing the modelling task than novice modellers. In addition, definitive patterns stored by a modeller also can be reused by other modellers. The method of reusing others' patterns resembles the way in which one person reuses another's experience to solve similar problems. In fact, this method of reusing past experience is very common among programmers. Very rarely does software system development begin from scratch [Pot93]. Many programmers have experience of reusing parts of previously developed program codes by pasting them into a new context with or without modification. It should be stressed that reusing program codes should not be regarded as simply reusing certain functionalities embedded in the codes; rather it involves the reuse of the programmer's past experience.

By means of the virtual agent concept and the generalisation-particularisation process, the modeller can create GOs and reuse their definitive patterns to reduce the size and complexity of a system. This benefit has been found in several practical case studies. For example, in an electronic circuit laboratory project for education, a GO called 'a painted button' is created and its definitive pattern is reused to create a further 150 similar buttons for storing diverse circuit graphs [Dor98, She98]. Another example is a classroom simulation project in which its source codes are rewritten by means of adaptable reuse proposed here (detailed in the next chapter). Each project reduces the total size of the model by more than 60%. Similarly, the case study of the simulation of a railway accident, discussed in the next chapter, also shows how a GO named 'a train' can be created and its definitive pattern reused to create a number of trains in the animation (see Appendix 6-B for more details).

Like the GO notion, one of the most important notions for reuse in software development is that of abstract data types (ADTs). The ADT is a very fundamental notion for modern, for example, object-oriented programming [Boo94]. In contrast to the standard data types provided by a programming language, an ADT is a programmer-defined data type whose logical behaviour is defined by a set of values and a set of operations on those values [Azm88, Cle86, Ber94, DW96, Wei99].

The most important notion behind ADTs is that of 'abstraction'. This term refers to a process that discards many details and emphasises only the 'main features' of interest at a particular 'level of concern'. It is used in many areas of computer science to reduce the complexity of tasks to a manageable level. C. Hoare suggested that "abstraction arises from a recognition of similarities between certain objects, situations or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences" [DDH72, p. 83, quoted from [Boo94, p.41]].

The essential idea behind ADTs is to separate the specification of an ADT from its implementation; hence, an ADT is a mathematical model [Azm88]. Nowhere in the definition of an ADT is there any description of how the set of values is represented and the set of operations is implemented. Encapsulation of this nature allows a separation of concerns: the user of ADTs can use the data type, but does not need to know how the data type is implemented. The specification of an ADT becomes the sole interface for both the people writing applications and the people who implement the abstract data type in a computer program.

Although both are based on the '*recognition of similarities*', ADTs and GOs are very different, as can be seen from the summary in Table 5-1. The most obvious difference is that an ADT is abstraction-based, but a GO is experience-oriented. As explained above, abstraction leads to the separation of specification from implementation.

On the one hand, this separation has the advantage of facilitating reuse, since the specification of an ADT can be used over and over again, and its implementation details can be considered only once. Moreover, once any value or operation in the specification of an ADT is declared, its associated implementation can also be reused without worrying about its details. On the other hand, such a strong separation of concerns means that, once a specification is given, only *prescribed* values and operations are available for reuse. Any attempt to adapt prescribed elements, including values and operations, or to access unspecified elements, becomes problematic [BE94, McG92, OS93]. Although this strong typing[13] feature makes programs manageable, easy to debug and more effective, it also to some extent discourages reuse [Ous98]. In these contexts, the reuse of ADTs can be viewed as creating instances from standard components whose descriptions are well-defined and unchangeable[14]. In other words, reuse can only reach the level of instantiation.

	GOs	**ADTs**
Typing	Typeless	Strong/weak
Form	Experience-oriented	Abstraction-based
Purpose	Reuse a definitive pattern for new observables; adaptable reuse	Separate specification from implementation; instantiate prescribed components
Character	Context-dependent	Context-free
Defined time	Prior/Run time	Prior
Usage	Situated, adaptable	Inflexible

Table 5-1. A comparison of GOs and ADTs

[13]The term "typing" is used to "refer to the degree to which the meaning of information is specified in advance of its use" [Ous98].

[14]In object-oriented programming, an alternative to reusing well-defined components is to reuse through inheritance, meaning that it is possible to overwrite the inherited description [GHJV95]. Though the latter is increasingly common in object-oriented programming, its suitability is controversial [GHJV95, Ous98, Som95]. Note that even when overwriting is allowed in an Object-oriented context, it still has to be done before run time.

In dtkeden, a GO is created to reflect the practical experience obtained in describing certain observables with the same characteristics. Its definitive pattern is not derived by discarding details, which – according to the concept of ADT – should be ignored prior to the process of abstraction: instead, it emerges from recognised similarities. For example, after describing a number of buttons, common features, such as their appearance and the action triggered by clicking the mouse left button, could be recognised and grouped together as the definitive pattern of a GO called 'an action button'. In this sense, the definitive pattern is shaped and generated through the practical, situated experience of the modeller in recognising these similarities between buttons.

No separation of specification and implementation concerns is possible in the context of such reuse in EM. No so-called specification can be given before the emergence of this practical experience, and no implementation details can be ignored during the generalisation process that leads to a GO, that is, by the work practice of 'recognising similarities'. This is because a GO is created in a situated fashion and on the basis of practical experience. This perspective on reuse is consistent with the concepts of 'experience in action' [Bur91, Hen96, LW91] and of Piaget's 'knowing-by-doing' [diS88, Puf88].

In summary, the mechanism of particularisation in dtkeden is to a large extent a form of reuse of the experience implicitly embedded in the description of a GO[15]. After generalising recognised similarities to the definitive pattern of a GO, the modeller is able to reuse the definitive pattern in the same way as experience is reused in everyday life when similar situations appear. Particularising the context of a GO to the context of an observable – specifying the virtual agent and including the definitive pattern into dtkeden – can be viewed as connecting past experience with the current situation.

[15]This is not to claim that definitive patterns exactly represent the experience obtained from the generalisation process. After all, experience cannot be totally represented in the form of language [Haml78, Jam96].

Localising the definitive script, in effect, can be regarded as a way of embodying reused experience into the computer model. The generalisation-particularisation process proposed here for adaptable reuse is, to a great extent, in line with the experiential learning process [Bur91] and the situated learning process [LW91]. The latter emphasises learning from practical experience and then applying it to future actions. Both concepts have been widely used in applied science, for example, in nursing and in training apprentices in work practices.

Appendix 5-A: The use of LSD notation

There is support for the LSD notation at the S-node in dtkeden when it is operating in 'normal mode'. Scripts in the LSD notation define agents' access privileges to observables, and a set of special dtkeden procedures is available to perform the same functions. Only some parts of the LSD notation are implemented in dtkeden - *oracle*, *handle* and *state*. Scripts in the LSD notation should start with '%lsd' and end with the name of another notation, such as '%eden'. The syntax is as follows:

agent *agentName*
oracle *observableName*[, *observableName*]
handle *observableName*[, *observableName*]
state *observableName*[, *observableName*]
remove *LSDType observableName*[, *observableName*]

The agent statement associates all the following statements with agent *agentName*. The oracle, handle and state observable lists contain observable names associated with the privileges to access the observables by *agentName*. The following example illustrates how to use the LSD notation.

```
%lsd            <----- declare the use of LSD notation

agent  xxx      <----- declare the agent name
oracle  a, b, c <----- add to agent xxx the oracle agency for observables a, b, c.
handle a, m     <----- add to agent xxx the handle agency on observables a, m.
remove oracle b <----- remove from agent xxx the oracle agency for observable b.

agent yyy       <----- declare a new agent name
oracle m, c     <----- add to agent yyy the oracle agency for observables m, c.
handle b, m     <----- add to agent yyy the handle agency for observables b, m.

%eden           <----- back to EDEN
```

After receiving the above description, the LSDagent creates the links between agents and observables. The LSD description can then be checked from the 'View LSD Description' subitem in the pop down menu of the 'View' item of the server's input window menu.

There is another way to configure the link between an agent and an observable. The commands 'addAgency', 'removeAgency' and 'checkAgency' can be used dynamically to manage the access privileges of an agent. Their syntax as follows:

addAgency("*agentName*", "*LSDType*", "*observableName*");
example: addAgency("yyy", "oracle", "b");

removeAgency("*agentName*", "*LSDType*", "*observableName*");
example: removeAgency("yyy", "handle", "m");

checkAgency("*agentName*", "*LSDType*", "*observableName*");
example: checkAgency("xxx", "handle", "w");

In these syntactic forms, the parameter *LSDType* is one of three basic keywords in the LSD Notation: oracle, handle and state. The first two commands are used as procedures in Eden, and the last command is used as a function returning a value of TRUE or FALSE. If TRUE (actually an integer of value 1) is returned by the command checkAgency, it means that the specified agent agentName has the identified privilege LSDType for the specified observable observableName. If the agent does not have this privilege, a FALSE (0) value is returned.

The current state of the S-node's interaction modes can be examined and changed via commands in the dtkeden input window. For the management of agents' privileges in 'normal mode', the LSD Agent in dtkeden creates and maintains three lists for each observable. For example, if the list of *oracle* privileges for an observable mmm is [xxx, yyy], then both agents xxx and yyy have the *oracle* privilege for the observable mmm. These three lists can be checked by using the function symboldetail("*mmm*"), which returns the privilege details associated with observable *mmm*. The format of the EDEN list returned is:

[*mmm*,*type*,*defn*,*l1*,*l2*,[*oracle_agents*],[*handle_agents*],[*state_agents*]]

example:

```
writeln(symboldetail("a"));
[a,formula,b+c;,[b,c],[],[EVERYONE],[sun,carters],[EVERYONE]]
```

For any observable in any state, an *oracle_agents* list, *handle_agents* list or *state_agents* list is either:

- empty - [] - no agent has the associated privilege;
- contains special agent 'EVERYONE' - [EVERYONE] - every agent has the associated privilege;
- contains a list of agent names (not including EVERYONE) - [sun,carters] - with the associated privilege.

When creating a new observable, dtkeden refers to a system variable called 'EveryOneAllowed' in order to give the default access privilege for each new definition of an observable. The two settings are:

EveryOneAllowed = TRUE;

In this case, a default agent name called 'EVERYONE' will be set for every new definition, and the LSD Agent will automatically grant every agent open access privileges to redefine and observe all newly created observables.

EveryOneAllowed = FALSE;

In this case, no agent has any access privileges to observe or redefine the observable.

Appendix 5-B: Virtual Agents

The concept of a virtual agent is motivated by the treatment of observables in the private interaction mode in dtkeden (see Section 5.3). In this interaction mode, a definition is introduced at an A-node is transmitted to the S-node in a form that is syntactical modified to its originating agent. For example, if agent X introduces a definition 'a is b + c', a new definition 'X_a is X_b + X_c' is generated at the S-node.

The virtual agent mechanism allows the superagent at the dtkeden server to introduce definitions in a context *as if* they were being generated by an agent in a similar fashion. The mechanism can be used in any interaction mode in dtkeden. If no virtual agent context for definition has been declared, all definitions are in the *root context*. In dtkeden, a virtual agent is declared by a symbol containing two characters. This symbol is followed by an agent's name that can be specified explicitly by a string constant (an *explicit* declaration) or implicitly by a string variable (an *implicit* declaration). In the former case, dtkeden will use the string constant as the current virtual agent's name to establish a context associated with this name, but in the later case the content of the given string variable will be used. If a symbol is not followed by an agent name, it indicates a reset of context to the root context. According to the given symbol, localisation is performed by appending the virtual agent's name to each variable identified in a postfix or prefix form. So far, five ways to declare a virtual agent have been devised in dtkeden. Table 5-B shows the use of these various methods to localise the definition 'a is b+c'.

The last symbol shown in Table 5-B does not cause dtkeden to shift the current context to the specified agent's name, so its localised definition is the same as the original definition. Normally, it is used for agency checking by the LSDagent embedded in dtkeden, to examine whether the named agent has appropriate access privileges to modify these variables in the root context. In this example, the LSDagent will check if the declared agent has the access privilege *handle* on the observable *a*.

In addition, the symbol '~' is available to reference the root context from another context, that is, it can be used to declare a global observable. For example, the definition '~a is ~b+~c' will be localised as 'a is b+c', whichever context is used. Furthermore, in the current version of dtkeden, the virtual agent can be applied to the Eden, DoNaLD and Scout notations.

Symbol	Declaring a virtual agent	Type of declaration of the virtual agent	Localised definition
>>	>>X	Explicit (in prefix form)	X_a is X_b+X_c
><	X = "x" ><X	Implicit (in prefix form)	x_a is x_b+x_c
<>	<>X	Explicit (in postfix form)	a_X is b_X+c_X
<<	X = "x" <<X	Implicit (in postfix form)	a_x is b_x+c_x
>~	>~X	The context that is associated with the virtual agent 'X' is declared, but localisation is not invoked	a is b+c

Table 5-B. Different ways to declare a virtual agent