Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

DEVELOPING A GRAPHICAL USER INTERFACE FOR DISCRETE EVENT SIMULATION

Hamad I. Odhabi
Ray J. Paul
Robert D. Macredie

Center for Applied Simulation Modelling (CASM)
Department of Information Systems and Computing
Brunel University
Uxbridge, Middlesex UB8 3PH, UNITED KINGDOM

ABSTRACT

A key concern for the area of discrete event simulation modelling is to encourage its adoption and use by non-specialists. To achieve this it is important that we focus on developing techniques and tools that can be used easily by people from outside the simulation modelling community. This paper explores the usefulness of one particular simulation modelling technique, hierarchical activity cycle diagrams (H-ACDs). We suggest that though H-ACDs are useful to simulation specialists, they can often be overly complex and can discourage non-specialists. We propose and develop a simplified version of H-ACDs, which we call simplified hierarchical activity cycle diagrams (SH-ACDs). We then go on to discuss the development of a simple graphical user interface which is the front-end to a modelling environment which supports the development of SH-ACDs. We discuss the way that an interface of this type can offer those without experience in simulation modelling the opportunity to develop simulations of their problem domain. We suggest that the SH-ACDs in conjunction with a suitable graphical interface can offer a potentially significant contribution by promising to open up the field of simulation modelling to non-experts.

1 INTRODUCTION

Discrete event simulation modelling offers people the chance to develop an understanding of their problem domain by building up a simulation of the problem space in which they are interested. This paper will suggest that the potential of discrete event simulation modelling may not be realised because the techniques and tools that are central to the field have generally been developed for and by specialists in simulation modelling. In this paper we use 'simulation modelling' to implicitly refer to 'discrete event' rather than 'continuous' simulation modelling.

We will develop this idea by focusing on a particular technique, Activity Cycle Diagrams (ACDs), which has attracted widespread coverage in the simulation community and is widely used by researchers and developers alike. We will discuss ACDs and suggest that they are a limited formalisms which are not particularly effective for modelling complex system behaviour. Modelling complex behaviour using ACDs requires in-depth knowledge and experience of simulation. This has been a limiting factor in their use by non-specialists. These issues have been realised and to some extent addressed by the development of Hierarchical Activity Cycle Diagrams (H-ACDs) (Kienbaum and Paul 1994a; 1994b). We will briefly look at H-ACDs and suggest that whilst they overcome some of the problems of ACDs, they still have characteristic limitations which suggest to us that they may not significantly widen the user-base to include those without experience in simulation modelling.

The importance of developing tools and techniques with non-specialist users in mind is the focus of this paper. We will suggest and explore two related ways in which we can move towards achieving the inclusion of non-specialists: (i) the development of simplified hierarchical activity cycle diagrams (SH-ACDs) which are less complicated and more usable through their decreased functionality; and (ii) the design of a graphical user interface through which non-specialists can develop SH-ACDs.

Before we move on to discuss ACDs and their derivatives (H-ACDs and our SH-ACDs), we will provide an overview of simulation modelling. We will use this overview to highlight why simulation modelling can be inaccessible to non-specialist users, why this inaccessibility impacts on the development and acceptance of simulation modelling, and to provide a broad justification for the developments that we report later in this paper.

2 SIMULATION MODELLING: AN OVERVIEW

One common approach to simulation modelling is for the modeller to develop a detailed model to describe the problem space in which we are interested. This model can then be run to assess its behaviour in particular circumstances, i.e. when particular objects and entities in the model are given particular values. Running the model provides a simulation of the problem in which we are interested.

There are many levels at which the development, and subsequent running, of a simulation model can be viewed as problematic for those without an in-depth understanding of simulation modelling.

One problem arises when we consider the steps that generally constitute the development of a simulation model. These steps may involve problem definition, abstracting and modelling the problem, developing a suitable representation of the model, running the model, testing the model, and analysing any results which arise from these steps. Following these steps to develop a complex simulation model may be extremely time consuming for the practised developer, let alone those who may be unfamiliar with the ideas involved in simulation modelling.

A more specific problem is that the model is often written in some computer programming language, expressed through a suitable program. In our experience, it is unlikely that the model will suitably capture the problem space at the first attempt. The model developer (or modeller) may have to significantly modify the program code that represents the model. To both develop and refine the model, the modeller will usually require a, potentially in-depth, understanding of the programming language. This precludes many people from outside the simulation modelling community.

Researchers in the simulation community have recognised the difficulties that exist for developers of simulation models generally, and those with little familiarity of simulation modelling specifically (Paul and Balmer 1993). In the following sections we will discuss a well established approach to developing a representation of the problem: activity cycle diagrams (ACDs). We will then suggest a simplified version of this approach which may, in conjunction with suitable computer-based tools to support its use, help address some of the problems facing non-specialists interested in developing simulation models.

3 ACTIVITY CYCLE DIAGRAMS (ACDS) AND HIERARCHICAL ACDS (H-ACDS)

Activity Cycle Diagrams (ACDs) are based on Tocher's (1963) idea of stochastic gearwheels. We can use ACDs to describe a problem space by specifying the objects and entities that exist in our problem space, and describing the states that the entities are in at any given time. These states can typically be: the entity is idle; the entity is in a queue (either a real queue in the problem space or a notional one defined to model a delay in the problem space); the entity is active; or the entity is engaged with other entities in some times consuming activity.

Though ACDs are powerful and have been, and continue to be, used to model problem spaces by the simulation modelling community, the ACD approach does have limitations. ACDs are used in simulation modelling to provide a simplified formalism of a potentially complex problem (Pidd 1998). This can cause difficulties as ACDs may not provide the modeller with the necessary representations to capture their problem adequately. We can see this when we consider a simple example. We can imagine an entity whose behaviour will depend in some way on its attributes at a particular point during a simulation. Since ACDs do not allow us to represent the attributes of an entity, we are bound to encounter difficulties when using ACDs to model the problem space. These and other limitations of using ACDs to model the problem space have led to the development of variants of ACDs which attempt to address the limitations.

One such variant is the Hierarchical Activity Cycle Diagram (H-ACD) (Kienbaum and Paul 1994a; 1994b). H-ACDs provide the modeller with a wide range of symbols which can be used to describe the problem. Whilst H-ACDs can represent complex problems more completely that ACDs the trade-off that modellers encounter is that the resulting diagrams seem more complex than ACDs and can take considerably longer to develop (we will develop this point in the following section). This may be off-putting to those from outside the simulation modelling community and is unlikely to widen the user-base of simulation modelling.

4 SIMPLIFIED HIERARCHICAL ACTIVITY CYCLE DIAGRAMS (SH-ACDS)

In the remainder of this paper we will develop a simplified version of H-ACDs, which we call simplified hierarchical activity cycle diagrams (SH-ACDs). We will suggest that these SH-ACDs go at least some of the way to addressing the concerns over perceived complexity that we raised when discussing H-ACDs. We will look in some detail at the key points of H-ACDs that provide the foundation for SH-ACDs, and the main differences between the two. This will provide an introduction to SH-ACDs. We will then go on to suggest that they SH-ACDs are sufficiently powerful to capture complex problems. Finally, we will describe how SH-ACDs may be made accessible to non-specialists by the development of a graphical user interface (GUI)

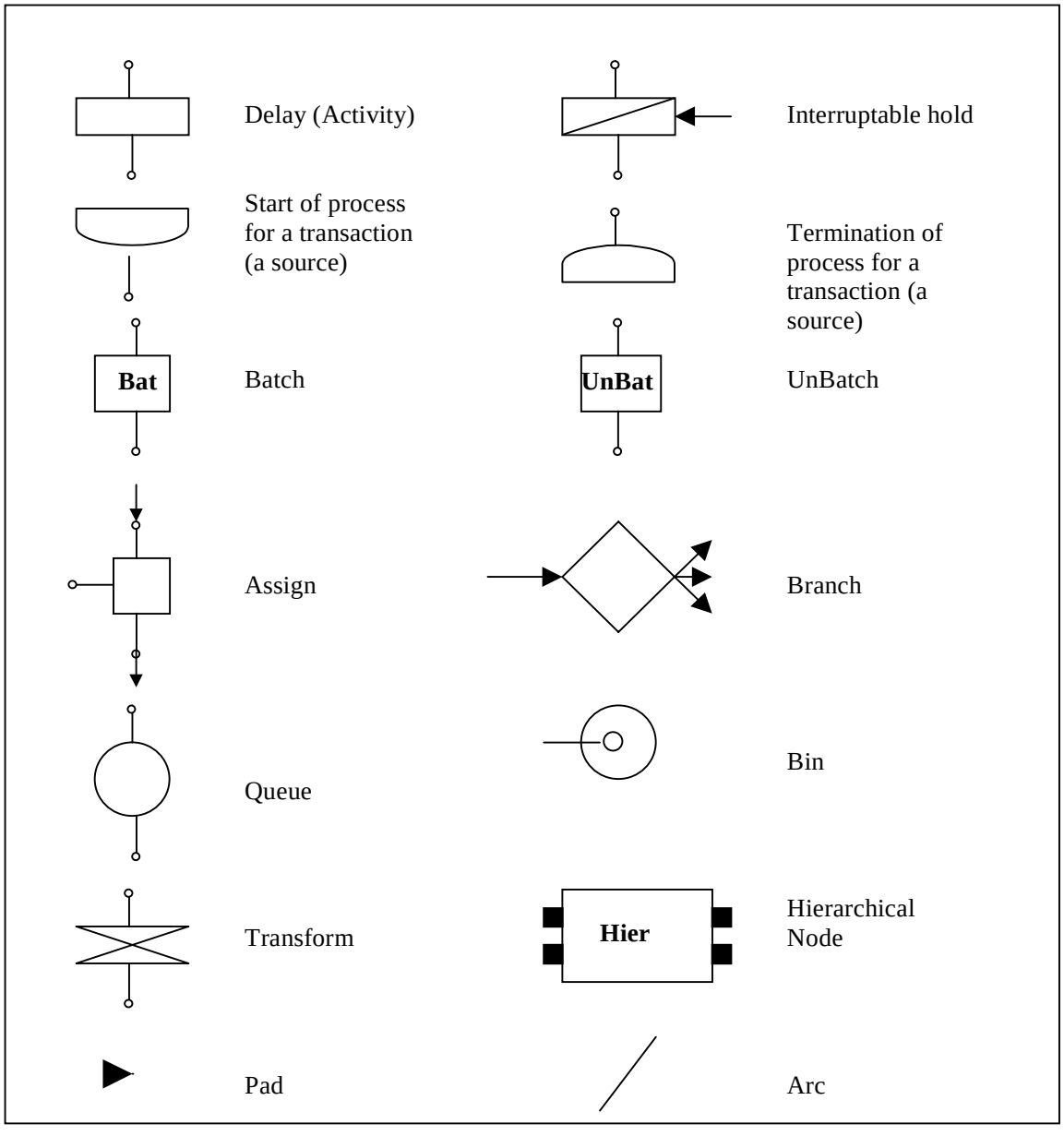

Figure 1: The Set of Process Nodes Supported by Simplified Hierarchical Activity Cycle Diagrams (SH-ACD)

through which users can develop SH-ACDs which model the problem in which they are interested.

Figure 1 shows the key process symbols that SH-ACDs offer to the modeller, and that are used to capture the problem space. These symbols are called atomic nodes; they represent the process types that SH-ACDs support.

4.1 Basic Commonalities Between H-ACD and SH-ACD

We will look here at the commonalities between H-ACDs and SH-ACDs, before moving on to the major differences. The common points can be categorised as follows:

1. Both H-ACDs and SH-ACDs support atomic nodes which represent different types of process, synchronisation, and queues that can be used to model the problem space.

2. The process nodes in both H-ACDs and SH-ACDs include source and sink nodes. They are used to represent the arrival and departure of an entity transaction in both specific parts of, and the overall, problem space being modelled.

3. Both H-ACDs and SH-ACDs support a 'transform' node which is used to describe a transformation process. Transformation refers to the change of an entity from one category to another.

4. The interruptable hold in both H-ACDs and SH-ACDs allows the modeller to build in interrupts which suspends an activity or process.

5. H-ACDs and SH-ACDs support the fundamental queue type.

6. The bin node is used by both H-ACDs and SH-ACDs to model the most general relationships between, for example, producers and consumers.

7. Both H-ACDs and SH-ACDs support hierarchical definitions of node types. This means that a 'father' class of node can be 'created'; subsequent 'child' nodes inherit the characteristics of the parent.

4.2 Basic Differences Between H-ACD and SH-ACD

To gain a better picture of SH-ACDs, we will now look at the major differences between SH-ACDs and H-ACDs, which can be summarised as follows:

(i) The 'assemble', 'request', 'delay', 'release', and 'disassemble' nodes that exist in H-ACDs are replaced in SH-ACDs by the 'activity node'. The motivation for this is that the use of five nodes to represent one activity makes the modelling more complex, tends to lead to the development of large models and can be very time consuming. The activity node is given attributes by the modeller such as its required number of resources and entities. These attributes govern the behaviour of the activity. For example, a particular activity may be modelled to only begin when a certain number of entities have 'entered' the activity.

(ii) The 'trigger' queue which exists in H-ACDs is not supported in SH-ACDs. Its function in instead shared between the 'branch' and 'activity' nodes. The branch is responsible for routing the entity correctly, in line with particular conditions which may relate to the entity type or its attributes.

(iii) H-ACDs support complex messaging mechanisms which explicitly pass information between nodes. This information reflects the on-going state of the simulation. In SH-ACDs these messages are hidden from the modeller. From the modeller's perspective, this suppresses unnecessary complexity.

(iv) Unlike H-ACDs, SH-ACDs do not limit resources to their initial values. Instead. resources may be added or removed from the model at runtime; the only stipulation being that these instances of these resource types should initially exist in the model. We believe that this more accurately reflects real-world problems, and reduces the burden on the modeller to fully specify the model at the outset.

(v) SH-ACDs simplify the use of 'pads' in defining the problems space, by offering a generic 'pad' type, rather than the four supported by H-ACDs.

(vi) Unlike H-ACDs, the 'arc' which connects nodes is provided with the name of the token that it carries. Upon completion, an activity will use this information to correctly route any relevant tokens.

This very brief review of the main commonalities and differences between H-ACDs and SH-ACDs has allowed us to point key points relating to SH-ACDs. Some of these points have helped us define a simpler environment for the modeller. In the following section we will expand on this theme by discussing the 'front-end', or interface, that we have developed to support the formulation of SH-ACDs. We will use this to demonstrate the mechanisms which support modellers in their development of SH-ACDs and suggest that the front-end represents a usable environment which may be helpful to non-specialist modellers.

5 A GRAPHICAL USER INTERFACE TO SUPPORT THE DEVELOPMENT OF SH-ACDS

A key motivation in our work is to develop a modelling environment that is accessible and usable by people from outside the simulation modelling field. The SH-ACD approach that we have developed from Kienbaum and Paul's (1994) H-ACDs provides the basic functionality that we feel a simulation modelling environment should offer. Crucially, this functionality has to be offered to the modeller in a way that supports their objectives: that offers them the chance to model their problem as easily, effectively, and quickly as possible.

A variety of front-ends to computer systems generally, and simulation modelling environments specifically, exist: from command line interfaces which rely on purely textual interaction through to graphical user interfaces which allow the user to graphically drive, or directly 'manipulate' (Shneiderman 1983; 1988) the interaction. Graphical user

interfaces are generally considered to provide support for novice users through their naturalness and intuitiveness.

The following sections of this paper will describe the graphical user interface (GUI) that we have developed to support the creation of SH-ACDs. The front-end and the underlying software to support the creation and manipulation of SH-ACDs represents a contained modelling environment which offers non-specialists the opportunity to graphically and interactively develop a model of their problem space. The interactive nature of the environment means that the user can be given real-time visual feedback as they construct their model.

The environment uses the iconic representations that we saw in figure 1 to represent components of the problem in developing the model. The use of icons which look, at least at some abstract level, like the objects and activities that they represent supports more natural interaction and helps make the system more usable than less natural interaction styles, such as that offered by command-line interfaces.

Each element of the model (icon) is displayed in the environment as a separate object with distinct properties. In the model, operations are performed on an object to change its state. The modeller uses the icons to build a representation, or model, of their problem space graphically on the screen. To support this the environment allows the modeller to select icons from a 'palette', connect them together, assign them attributes, and to generally manipulate the icons to define their problem space. When they have modelled the problem they can run the simulation to assess its behaviour.

The graphical interface that we have developed as a front-end to our modelling environment will be referred to in the remainder of this paper as SH-ACDGUI.

5.1 SH-ACDGUI: Core Functionality

In this section we will look in a little more detail at the main functionality that the modelling environment offers to the modeller through the front-end (SH-ACDGUI). We will discuss the general functions before looking at the ways in which the front-end supports interactive graphical specification of the model. The general functionality can be described in terms of the following components of the modelling environment:

(i) editor: the editor has five main components - the menu bar, the location field, the palette bar, the status bar, and the layout area. The menu bar is located at the top of the editor and contains a number of menus. The location field gives the user an indication of their position in the hierarchical model that they have developed to represent their problem space by showing the full name of the current topmost node. If the user is at the highest (or top) level, the name of the model will be shown. If the user entered a hierarchical node, the name of the model, followed by a colon, followed by the name of the node will be visible. The third area is the palette bar. The palette bar contains icons representing various kinds of objects that can be added to develop the model. The layout area contains an iconic representation of the topology of the model. It contains graphical model elements representing nodes, pads, and arcs, and visual cues such as background icons and text. The background icons and text have no bearing on a simulation; they are used only to enhance the visual appearance of a model. The last component of the editor is the status bar. The modelling environment sends error, warning, and trace messages to the status bar as the editing session or simulation progress to inform the user of the progress of their simulation.

(ii) nodes: nodes are, as we have already seen, shown in the environment as icons. Nodes represent processes or hierarchically arranged groups of processes in model and . These are implemented as two separate classes- process nodes and hierarchical nodes. The modelling environment has a node type which is a common ancestor, or base type, of these subtypes. A process node is a non-decomposable, or atomic, model element. Different subtypes of process node are used to model sources, delays, transformations, batch and unbatch processes, branches, assignments and sink processes. The modeller can quickly edit the parameters of a node by double-clicking on the relevant node icon and filling in the fields of the resulting dialogue box. A hierarchical node contains other nodes that may themselves be hierarchical nodes. There is no imposed limit of depth to which nodes can be nested.

(iii) pads: pads model input and output controllers for nodes to which they are attached. When a node has to route an entity in the simulation, it 'passes' the entity to one of its associated pads and requests that the pad routes it. Individual pads 'know' the node to which they are connected; and nodes maintain lists of pads that they own.

(iv) arcs: arcs are used to connect pads. Individual arcs know the pads that they connects and the entity that they are designed to carry. Pads keep lists of the arcs with which they are associated. Arcs do not delay, process, or transform entities; they merely indicate a connection between two pads.

(v) tokens: the token represents an entity moving through the simulation model. Tokens only exist as the simulation model is run. Tokens can model any kind of entity, depending on the problem domain being modelled. Examples may be: packets or frames in a communication network, parts on a factory floor, or customers in a bank. Our modelling environment actually implements the concept of a token through two objects: the token 'category'

and the token itself. The token category holds the parameters of the token, such as its name, the icon that represents it, its attributes. When the simulation model is run, the modelling environment creates appropriate tokens (in line with the model's logic) that flow through the model. Each token has an associated token category, and represents a single instance of the specific category. As the model runs, tokens which point back to the correct category are created (by the source node as we shall see in the following sections). Separating the token from its category means that individual tokens do not have to carry the name, icon name, attributes, and statistics request information; this information is only held in the token category pointed at by the token.

(vi) resources: the modelling environment uses 'resources' to model domain elements that constrain the model's performance. Examples of domain elements that might constrain performance are manpower, communication links, or processing elements. Tokens are required to wait at the relevant node until the resource requirements of an activity are available. For example, a particular process may require a particular minimum level of stock before it can begin. The stock is the resource on which the process is dependent. The process has to wait until the stock level is appropriate.

(vii) attributes: the modelling environment uses variables which are called global 'attributes' to carry information around the model. This information may be used to define particular aspects of the model's performance. For example, 'attributes' may be used to constrain the performance of the model by setting global constraints on activities, i.e. processes have to be suspended between certain hours.

We will now go on to look in more detail at the types of process nodes that are available in the modelling environment and the ways in which the front-end (SH-ACDGUI) supports their use by the modeller.

5.2 SH-ACDGUI: Process Types

The types of process nodes supported include 'source', 'queue', 'delay', 'sink', 'batch' and 'unbatch', 'transform', 'branch', and 'assign'. All of these types are derived from a process node. Many of these process are directly inherited and/or developed from those supported by Kienbaum and Paul's (1994) H-ACDs. The following are brief descriptions of the process types that are presented to the user through SH-ACDGUI:

(i) source nodes: the 'source' node is a process node that creates tokens of a particular token category. It knows how to own a signal generator and accept a signal from it. When the 'source' node receives the a message from the simulation manager instructing it to begin running the simulation model, the 'source' node performs the inherited processing and passes the message to its generator. It then resets initial numbers of tokens so the feedback given to the modeller through the account displayed on the node's icon is correct. The 'source' node's generator sends accept signal messages to the source node as the generator moves through time. The 'source' node uses the signal as a prompt to create a suitable number of tokens. Each time it receives a signal, the node draws a sample to determine the number of tokens to create.

(ii) delay nodes: the 'delay' node delays tokens for some amount of time before sending them on their way. Delays are required for particular activities in the model when, for example, activities have to wait for particular resource levels to be built-up. The 'delay' node itself is not capacity limited: it can process any number of tokens at the same time. The activity node's primary input pad passes an accept message to the delay node when it accepts a token from an upstream pad in the model. When the activity node receives a message from any node, it checks the required numbers of tokens in the previous nodes and the required resources which are idle in the system. When all the required tokens and resources are ready, the activity node sends messages to all the previous nodes to send the required tokens. The activity node then assembles the tokens, requests the appropriate resources, performs the inherited processing and implements the required delay.

(iii) sink nodes: the 'sink' node terminates tokens which arrive at it. It marks the location at which tokens exit the model. The 'sink' node is also allowed to terminate a simulation prematurely if a certain number of tokens exit the system at the node.

(iv) batch nodes: the 'batch' node batches a number of tokens into a single token. The batched tokens do not lose their identity, as they are merely added to a list of tokens carried by the resulting batch token. The node identifies the number of tokens to put into a batch, and the category of batch token. The 'batch' node assess whether or not it has a reference to a token category. If it does not, it generates an error message that prevents the simulation from being run. The node receives an accept message when a token enters its input pad. Batch tokens are produced when sufficient tokens in relevant categories exist at the node.

(vi) unbatch nodes: the 'unbatch' node ignores tokens that are not batch tokens. For a batch token - that is, a token carrying other tokens - the node empties and then disposes of the batch token, and sends each of the consistent tokens

on its way. The 'unbatch' node has no new parameters, and so needs no dialogue control or load and save behaviours.

(vii) transform nodes: the 'transform' node transforms a token of one type into another. The transformation is pre-defined by the modeller.

(viii) queue nodes: the 'queue' node delays tokens until a message has been received from the appropriate activity node. The queuing conditions are pre-defined by the modeller as part of the overall model logic. When the simulation is started the queue will initialise the pre-defined requirements and wait to receive a message from the related activity node. If there is no activity node connected to it, the 'queue' node sends the tokens straight through. When a token arrives at the 'queue' node, it will send a message to the relevant activity node informing it of how many tokens are queued.

(ix) branch nodes: the 'branch' node routes decisions in the simulation. These decisions reflect the modellers individual problem space. For example, the modeller may want to route tokens to different arcs, depending on the token name or its attributes.

(x) assign nodes: in most practical simulation models it is necessary to change the values of entities or system attributes at some time during the running of the model. The 'assign' node is used to perform this process.

5.3 SH-ACDGUI: Development Platform

Our modelling environment was developed using the MODSIM II language (MODSIM II 1992) and the SIMOBJECT library (SIMOBJECT 1994). The flexibility that they offer is critical to they environment. Of particular importance are the following provisions: concurrency, simulation time facilities, random sampling, animation, and graphics facilities. They also provide other facilities that support the implementation of our graphical user interface (SH-ACDGUI), including support for dialogue management and menus.

In this section we will look at the main features of MODSIM II and SIMOBJECT that underpin our modelling environment. We will not focus in detail on the implementation of the features. Rather we will be interested in the concepts that they represent. The main features that we make use of can be listed as follows:

(i) concurrency: concurrency allows a number of activities to occur simultaneously in the timescale of the simulation. Activities can operate autonomously or they can synchronise their operation.

(ii) simulation time: MODSIM II supports discrete event simulation. Through its modules it contains the definitions of all constructs needed to run process-driven simulations. The units of time used by the simulation are dimensionless, and they can represent whatever granularity of time is thought appropriate by the simulation modeller. It is up to the modeller to explicitly perform any unit conversions. Simulation time is automatically maintained by MODSIM and is reset each time that the simulation is run.

(iii) random sampling: in many cases, an object will have some behaviour that is best characterised by random sampling from a statistical distribution. SIMOBJECT supports a basic object type which is designed to model this behaviour. It is possible to establish a default distribution, draw samples, load and save parameters, format the distribution for presentation to the user, provide dialogue box interaction for the user to specify parameters, present a plot of the distribution, and manage a combo box to let the user pick from a list of available distributions. In most cases, it is necessary to let the user modify both the distribution type and its parameters via a dialogue.

(iv) animation: an animation manager is supported, which controls the amount of animation performed during the simulation. The manager does not actually perform the animation processing; that is up to the individual objects involved in the simulation. Instead, it presents the user with a single point of control over the animation processing.

(v) graphics: the graphics library manager provides a simple mechanism for getting an image from a graphics library file. It also provides a tool to allow the developer to easily build a combo box list containing icons suited for a particular application.

6 CONCLUSION

This paper has introduced a simplified form of H-ACDs, called SH-ACDs, and has briefly described the environment which we have developed to support their implementation. The focus of the paper has been on the importance of developing an interactive, graphically driven environment which supports the modeller and has the potential to be accessible to those without in-depth knowledge of simulation modelling. The work that has been reported in this paper is the first stage of wider research which will go on to assess the effectiveness of the modelling environment.

This will allow us to test our assertions that the front-end to the environment, SH-ACDGUI, is likely to reduce the time required to build a model by allowing an interactive graphical development which is more intuitive to the user.

We believe that this will allow the modeller to concentrate on understanding the modelling process rather than the tools that they are using to develop their model. We believe that this will help remove a major barrier to widespread use of simulation.

REFERENCES

Kienbaum, G. and R. J. Paul, (1994a). H-ACD: hierarchical activity cycle diagrams for object-oriented simulation modelling. *In the Proceedings of the Winter Simulation Conference* (IEEE Cat. No. 94CH35705), edited by Tew, J. D., Manivannan, M. S., Sadowski, D. A., Seila, A. F. (New York: USA).

Kienbaum, G. and R. J. Paul (1994b). H-ACDNET: An object-oriented graphical user interface for simulation modelling of manufacturing systems. *Simulation Practice and Theory*, 2: 141-157.

MODSIM II (1992). The Language for Object Oriented Programming (reference manual) CACI Products Company (La Jolla, CA.).

Paul, R. J. and D. W. Balmer (1993). *Simulation Modelling* (Lund, Sweden: Chartwell Bratt).

Pidd, M. (1998). *Computer Simulation in Management Science* (4rd edition) (Chichester, UK: John Wiley and Sons).

Shneiderman, B. (1983). Direct manipulation: a step beyond programming languages. *IEEE Computer*, August: 57-62.

Shneiderman, B. (1988). *Designing the Interface: Strategies for Effective Human-Computer Interaction* (Reading, Mass.: Addison-Wesley).

SIMOBJECT (1994). CACI Products Company (La Jolla, CA.).

Tocher, K. D. (1963). *The Art of Simulation* (London: English University Press).

AUTHOR BIOGRAPHIES

HAMAD I. ODHABI is a researcher in the Department of Information Systems and Computing, Brunel University. He received a B.Sc. degree in Physics from King Suad's University, Saudi Arabia in 1988, and he received an M.Sc. degree in Simulation Modelling from Brunel University in 1994.

RAY J. PAUL holds the first U.K. Chair in Simulation Modelling, at Brunel University. He previously taught Information Systems and Operational Research at the London School of Economics. He received a B.Sc. in Mathematics, and a M.Sc. and a Ph.D. in Operational Research from Hull University. He has published widely in book and paper form (two books, over 200 papers in journals, edited books and conference proceedings), mainly in the areas of the simulation modelling process and in software environments for simulation modelling. He has acted as a consultant for variety of United Kingdom Government departments, software companies, and commercial companies in the tobacco and oil industries.

ROBERT D. MACREDIE is a reader in the Department of Information Systems and Computing, Brunel University. He received a B.Sc. in Physics and Computer Science and a PhD in Computer Science from Hull University. His research interests are in human-computer interaction, simulation modelling, and virtual environments/virtual reality. He has published widely in these areas, and is also executive editor of the international journal *Virtual Reality: Research, Development and Applications.*