Component-Based Modelling and Simulation: A Case for Simulation Software Architecture

Ddembe Williams and Michael Kennedy
School of computing,
South Bank University
103 Borough Road,
London, SE 1 OAA, U.K,
Tel: 0171 815 7460 Fax: 0171 815
E-mail: d.williams@sbu.ac.uk; kennedms@sbu.ac.uk

Key words*: Components, Complex Systems, Object-Oriented Development Simulation Software, System Dynamics.*

Abstract

This paper reports the results of an initial study into the utility of the application of Component-Based Programming (CBP) techniques to the development of a System Dynamics (SD) simulation model. Many problems being examined by SD techniques are complex and thus time consuming. Object-oriented (OO) and visual simulation technology (VST) are becoming increasingly important in software development but, although OO component reuse in software engineering is about 10 years old, this concept is new to SD simulation software. SD simulation software has greatly improved in usability and user acceptance in recent years, although the modelling process is still tedious particularly for people new to SD. This is often due to a lack of experience in the conceptualisation process and in gaining understanding of the problem domain. The paper further suggests that there are potential benefits in using a CBP architecture in SD modelling as it leads to shorter development and simulation time, higher user acceptance of and greater confidence in the models developed.

1.0 Introduction

This paper reports results of an initial study into the utility of a component-based simulation environment. This describes our experiences with the application of CBP (using C++ Builder) to the development of a SD simulation model. Due to recent advances in object-oriented and visual simulation technology, they are becoming increasingly important (Balci et al, 1997; Brown, 1996; Umar, 1997) in software development.

Although traditional system dynamics simulation software has greatly improved in usability and user acceptance, the modelling process is still tedious particularly for people new to SD, due to their lack of experience in conceptual process and understanding of problem domain. The concept of reusable SD model components for visual simulation illustrates how a visual simulation model can be used to develop SD

model with no programming experience, and in a short development time. (Umar, 1997; Miller et al, 1998; Balci et al, 1998; Harrel and Hicks, 1998 and Mackulak et al, 1998).

Results of the study indicate that simulation software can utilise the benefits of Object-Oriented Programming. Component-based simulation software in SD has the potential of adding capabilities to maximise reuse of model parts or even whole system depending on complexity of the system in question.

The benefits identified in this study of using ActiveX and C++ component-based simulation in the software can lead to short development and simulation time higher user acceptance and confidence in system dynamics simulation models. Programming languages have evolved constantly over the last three decades and along with this evolution has come several programming styles aimed at reducing the cost and time of program development. The rest of the paper is divided in five sections. Section two describes the current simulation software available, while section 3 introduces the concept of component-based programming. Section four discusses our learning experience in using component-based simulation architecture and section five provides the summary and concluding remarks.

2.0 Current Simulation Software

The 1960s saw a great progress in the development of both analogue and digital computers and complex simulation studies became a reality. In the early days of computer based simulation, analogue computers were used for the simulation of continuous systems whilst digital computers were used for discrete event systems (Matko et al, 1992). Analogue computers were used to simulate continuous systems because they allowed the representation of continuous variables by continuous properties such as voltage and current (Pidd, 1992).

Current simulation software used in modelling systems that are predominantly continuous in nature are called SD software packages. The term “simulation software” is used in a broad sense to describe simulation languages and software packages or "simulators".

2.1 Simulators

Law and Kelton, (1991) describes a simulator as a computer package that allows the simulation of a system contained in a specific class of systems with little or no programming. Schmidt, (1986) identifies simulators as possessing a main program and a library of subroutines which the user introduces to carry out desired function in the simulation program. We contend that simulator are extendable if the user knows the base language well enough to create new subroutines which can be appended to the simulators library of subroutines.

2.2 System Dynamics Simulation Software

Simulation software designed for the developments of SD simulation models offer some common approaches to model building. In general, these tools enable model development through graphical specification of required relationships among variables (STELLA/ithink, Vensim, Powersim), or by the implicit writing of equations in text editors (COSMOS & COSMIC, Dysmap2, Dynamo). They also provide built-in functions which covers a wide range of simulation, mathematical and logical, statistical and analytical tools in the form of tables, graphs, animation, flow charts, or reports which explains simulation results. Some offer additional functionality for model sensitivity testing and optimisation (Coyle, 1996).

2.3 Object-Oriented Programming

The Object-Oriented paradigm views a program as a collection of discrete objects that are self-contained data structures and methods that interact with other objects. It provides a way of dividing a program into modules by using objects as building block. The concept of reuse is central to object-oriented programming and this is achieved through inheritance. Whilst there are pure object-oriented languages such as Eiffel and Smalltalk, there are also hybrid-languages such as C++ which provides all the crucial elements for applying the object-oriented paradigm, and there are others which are categorised as object based because they contain some aspect of encapsulation inside objects that can be created from a set of existing classes, but does not provide the mechanism for creating new classes. An example of an object-based language is Ada 95.

3.0 Component-Based Programming (CBL)

The concept of reuse is central to CBP is a visual programming style that involves the use of independent, pre-fabricated software units called components, and these components are objects which encapsulates data, method and events, ready for use in an application (Borland, 1997). CBP is implemented in Integrated Development Environments (IDE), based on programming languages, which are Object-Oriented in nature. It offers a simplified way of programming through its collection of pre-fabricated components, and other programming features, which typifies an IDE. These environments are also recognised for their capacity for rapid application development (RAD). CBP is not entirely free of code writing, as there is usually the need to augment the behaviour or extend the functionality of a component to suit application needs. It is also the case that the incorporation and use of these components sometimes necessitate additions to the syntax of the programming language on which they are based. The technology of component creation appears to fulfil the software industry's need for true reusability as described by Cox (1990). The standards in place, such as the Object Management Group's [OMG] Common Object Request Broker Architecture [CORBA], (which provides common interfaces and descriptions for objects), and Microsoft's Component Object Model (COM) which provides a specification for using objects

produced by various vendors within a single application).

By adopting a CBP approach in the development of a SD simulation model, this work aims to identify any improvements it may offer over the normal style of programming which would make the programming approach to computer simulation modelling comparable to, or more advantageous than the use of a Simulator. For this purpose, C++ Builder has been chosen as the implementation tool.

3.1 Overview to C++ Builder

Borland International describes the C++ Builder as a visual programming environment where 32-bit Windows applications can be designed, developed and debugged. The style of programming supported by C++ Builder is widely known as CBP. Telles (1997) describes it as a true Rapid Application Development (RAD) tool for the C++ Windows programming world. The OO concept of reuse is central to CBP. C++ Builder offers programmers a dual approach to programming using the C++ language. It offers the opportunity to program within an Integrated Development Environment that allows programs to be written, edited, compiled and linked, all within a single application, and it also offers CBP in which components are used in applications to ease the programming task and cut down on development time. Standards established for the creation of components require that they are universal and transferable from system to system regardless of language. In order to allow for the creation of components which may be used on other platforms, and facilitate the use and manipulation of components from other platforms in C++ Builder applications, additions were made to the C++ syntax. Figure. 1 shows an editor, a form, the object inspector, the component palette and the tool bar in the start up screen of C++ Builder.

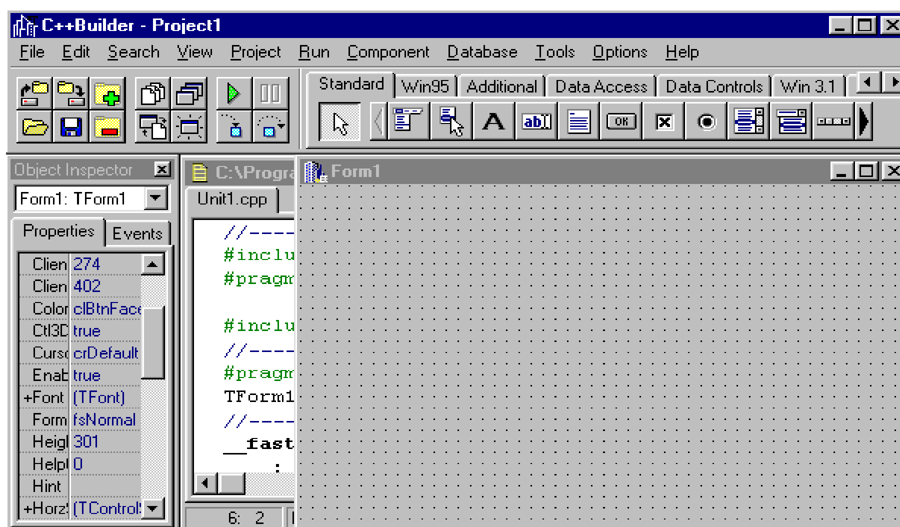

Figure 1. The Start up Screen of C++ Builder

3.2 Components in C++ Builder

The C++ Builder describes a component as an object in code. Visual Component

Library (VCL) contains different types of objects, some of which are not components. Components in C++ Builder are identified as any class, which is directly descended from the TComponent class. Objects, which are not components, are derived directly from TObject, the ancestor class at the top of the VCL hierarchy.

VCL is a good example of the use of inheritance, in object-oriented terms. TComponent, the ancestor of all components in the VCL provides the minimal properties and events necessary for a component to work in the C++ Builder environment. Other base classes, which descend from it, have specialised capabilities, which are present in all classes, derived from them. For example, all edit-box and memo controls are derived from the TCustomEdit base class that encapsulates the fundamental behaviour common to all components, which edit text.

One significant characteristic of all components is that they are visual and can be manipulated at design time. However, whilst some components remain visible at runtime, some do not play any visible role in an application at runtime because they are program elements which act either as a placeholder, or an interface for the manipulation of underlying data. For example, DataSource is a non-visual (at runtime) component that provides a conduit between a dataset and data-aware controls that enable the display, navigation and editing of underlying data in the dataset. Non-visual components such as these are derived directly from TComponent whilst visual components such as Edit Boxes (also know as controls) are derived from TControl, a specialised base class derived from TComponent. An another characteristic of CBP is the universality advocated for component use and distribution. Some of the components are written in Pascal whilst some are outright in ActiveX controls. C++ Builder also supports the development, installation and use of new components in its environment.

3.3 Using C++ Builder Components

Using C++ Builder Components involve the selection of a component from the component palette or from the main menu under View/Component List, dropping it unto a Form, setting its properties in the Object Inspector and writing an event handler (code) in the code editor for any event (listed in the Object Inspector) which the component responds to. To aid flexibility in their application, new components can be derived from existing classes to extend functionality, get rid of properties which are not required, override methods, or change standard default values. However, this derivation cannot be made directly from the component class as existing properties and method cannot be changed. A base class of the type from which the component is derived would have to be used. For example, if an Edit Box with additional functionality other than that provided is required, a new class should not be derived from the TEdit class of components; it must be derived from the TCustomEdit base class.

In addition to the VCL, C++ Builder has a standard library of functions and templates that form a large part of applications developed in the C++ and some example of their use will be demonstrated in the next section.

3.4 Developing SD Models with C++ Builder

In programming a simulation model, the modelling process involve the direct writing of equations in code form as opposed to the physical assembly of building blocks when using certain simulation package. However, it is possible to visually model both the equation and simulation interface given the availability of suitable components. Currently, C++ Builder has an extensive range of components, some of which are dedicated to specific types of application. An extensive collection of components is dedicated to use in database development and manipulation, and for Internet applications. There is however no components specifically developed for use in SD simulation modelling.

As the model variables in SD (except for constants) are dependent on equations which makes use of other variables to get their value, it is necessary to initialise the declared variables (placing them in the right order of execution) with the appropriate value or equation in the Forms constructor, to enable their use in subsequent calculations. The initialisation process involves assigning initial values to stocks and constants, and equations to flows and converters to set their values. Once initialisation is achieved, run time equation is written to set the cumulative value of stock variables. The next step is to establish a link to a graphics object that would display the result of calculations during model simulation and these forms a major aspect of the simulation interface building process.

3.5 Simulation Interface Programming

Graphs are the best means for displaying simulation results. C++ Builder offers two graph components on the ActiveX page of the component palette but initial explorations showed them to be unsuitable for use due to a lack of adequate documentation for their application, and the fact that their use within an application requires a major deployment of supporting files which must be installed and registered on the client's machine before application installation can take place. Nevertheless, it is possible to procure independent components from other source and install them into C++ Builder.

To use the procured graph component for simulation purposes, it important to override and re-write methods which set graph properties such as lines, tick marks, line points, axes values, grids, etc., For example, the method below sets the value of the x-axis grid by first checking to see if the value has changed, and if it has, the new values is given to the FXGrid data member which holds the x-axis grid value, and the standard repaint function is called to repaint the grid.

```
Void __fastcall TDGraph::SetXGrid(bool xGrid)
{
            if (xGrid !=FXGrid)
        {
            FXGrid = xGrid;
            Repaint();
          }
}
```

As data members in C++ cannot be accessed directly, the VCL provides properties, which are made available for setting in the Object Inspector by declaring them as, **published**, or called only as methods by declaring them as **public**. The “read” and “write” functions used to accomplish this task is demonstrated below.

```
private:
   bool   FXGrid;
public:
  __property bool XGrid = { read=FXGrid, write=SetXGrid, default=true };
```

An important property of the graph, which required special attention in its creation because it forms a link to the model's data, is the line property. Lines are created by linking a series of X and Y points on the drawing canvas and considering that the number of points generated by an equation during a simulation cannot be pre-determined at design time, functions provided by the C++ Standard Template Library which reserves memory and have the capability to dynamically increase as needed. This concept is demonstrated below.

```
void __fastcall TDGraph::SetNumberOfLines( int nLines )
{
     FXPoints.reserve( nLines );
     FYPoints.reserve( nLines );
     FLineColors.reserve( nLines );
     for ( int i=0; i<nLines; ++i )
     FLineColors[ i ] = clBlack;
     FNumberOfLines = nLines;
}
```

When changes are made to the underlying data through the Tab Dialogue, clicking on the run button on the tool bar can refresh the graph. Figure 2 shows the simulation output graph defined in the code above.

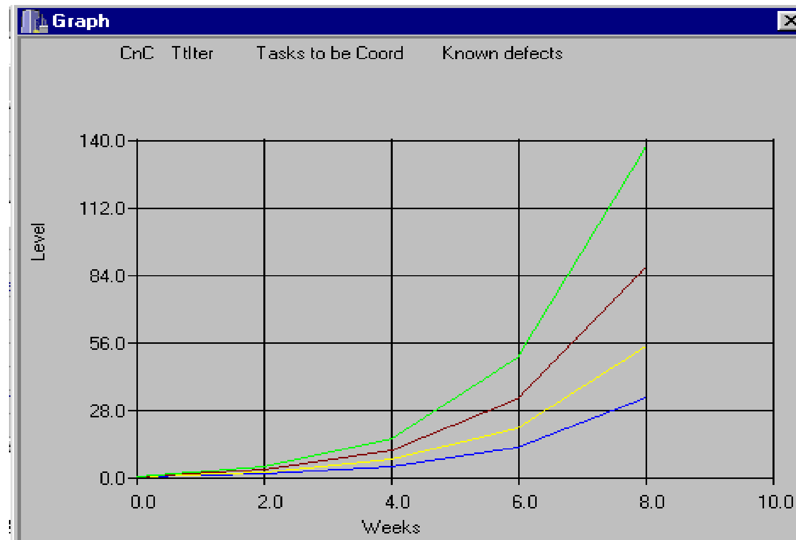

Figure 2. Simulation Output Graph

In the main application, an instance of the graph component is added to the TForm's header file to enable its use, and properties such as the number of axes ticks, decimal places, axes values are set for model use. For example, when a simulation time is specified by the user, the change has to reflect in the x-axis tick marks, which displays the time gradation. To enable this, the maximum time is divided by the DT (unit of time) specified and the value is assigned to the NumberXTicks property of the graph which redraws itself as the values change. Through this kind of assignment, model data such as the result of calculations are set into the graph's line property.

3.6 Interface to Model Data

Once the underlying implementation is complete, an interface that facilitates data input for onward assignment to graph properties is created using a collection of components. The interface is a tab dialogue created with the Page Control component and it is designed to group related graph properties together, separate from model properties. Components such as the Panel, Label and Bevel are used mainly to set the appearance of the dialogue box whilst Edit Boxes serve as the major source of user input. Radio Buttons and Check Boxes are used to indicate user choices and ComboBoxes present a list of items for users to choose from. To accept input from a component such as an edit box and assign the value to a property, it is necessary to write code as shown below.

```
double xdecimal  = atof(Form3->FilterEdit1->Text.c_str());

nGraph->NumXDecimals = xdecimal;
```

In the above code, a variable of the type **double** is used to accept and store the value of the text, which has been input into FilterEdit box1. Because of the generic AnsiString

type used throughout the VCL, it is necessary to convert AnsiString to character by appending the c_str() function to the end of the FilterEdit box's Text property. In addition to this, the input text, which is really a number, must be further converted into the data type (**double**) of the NumXDecimal property of the graph object before its assignment. Otherwise the direct use of input data produces some very strange results. This conversion is performed using the "atof" function from the stdlib.h / math.h library. Figure 3 shows the multi-line Model Properties Tab Dialogue.

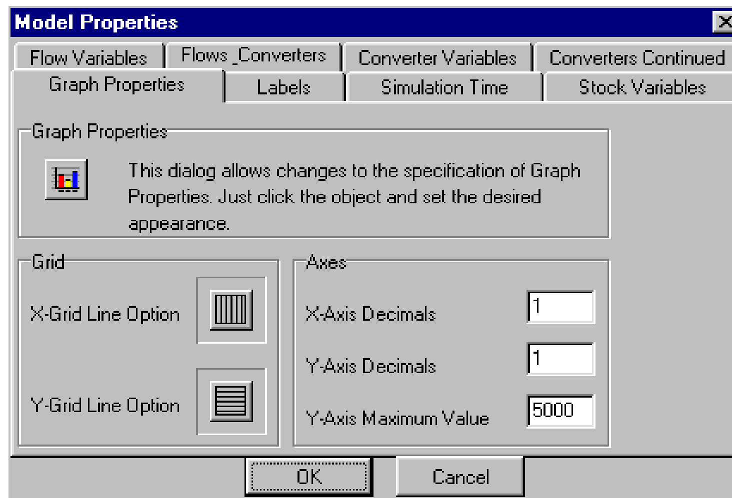

Figure 3. Model Properties Tab Dialogue

The main interface to the application consists of a Main Menu and a tool bar made up of Speed Buttons that are used mainly as shortcuts to the execution of various functions within the application. Creating shortcuts to main menu item using Speed Buttons involve assigning the event handler of the original menu item, to the On Click event of the shortcut button in the Object Inspector.

4.0 Learning Outcomes

In this section we report our experiences gained with modelling with components.

4.1 Learning the Tools

Most of SD simulators provide a clear and well-documented online help facility, which is very useful in learning its application to model development. The help facility makes extensive use of illustrations to explain functions of various tool features and this visual aid helps the learning process immensely. Learning to use C++ Builder for CBP is somewhat tedious. It is necessary to become acquainted with the components available, their properties, methods and events, and also learn how to use them within an application. These components and the vast number of object classes available within the VCL are documented in volumes of manuals and in the online help facility. For an

experienced C++ programmer, learning and using C++ Builder may be a fairly simple task but for a user with little previous programming experience, the learning process may be time consuming. In our experience, we have found that there is a steeper learning curve with CBP, however in our opinion the potential advantages in terms of functionality and flexibility of using CBP approach are far greater than the use of simulators.

4.2 Skills Level Required

In using current simulators, presuming the modeller has knowledge of the fundamentals of System Dynamics simulation modelling, no special skill (programming or otherwise), is required to successfully use the tool, other than to learn the specifics of features and functionality offered by the package. With CBP using C++ Builder, there is still the need to write some code even though the level of functionality, which has been built into the components, vastly reduces the level of effort and skill needed. However, programming a SD simulation model without components specifically provided for such use still require real technical skills and the benefits offered by the Component-Based approach in other situations therefore does not apply.

4.3 Accessibility and Portability

Programming languages and tools are more widely available in many organisations because of their prevalent use in the development of a variety of applications. For this reason, a developer is more likely to have access to a programming tool such as C++ Builder than to a specialised simulator. When a model is developed with a programming tool such as C++ Builder, the model can be installed for use on a wide variety of machines as an independent application. With the use of simulator however, the final model can only be ported for use to a system which has the simulator software, and moving it on to another platform would requires the additional effort and cost needed to translate it.

4.4 Relevance of CBP to Simulation Software Architecture

The modelling framework provided by graphical SD simulators directly supports the dynamics elements: Levels, Rates and Auxiliaries, and it also enables the specification of relationships between these elements through links created by connector object. C++ Builder being a generic programming tool that is not geared towards any specific application development does not offer features which directly support the fundamental principles of SD modelling. However, the C++ language, has constructs, which can be used effectively to represent the SD elements.

Using simulators is a fairly simple and straightforward process. However, building large and complex models can be very difficult for model owners to unravel and navigating the web of connectors between numerous model entities makes them difficult to understand. Time is wasted using the building blocks whilst dealing with the problems

of the model. One very useful feature of simulators is that they provides underlying support in the specification of equations, arranging them in the necessary order of execution and creating part of the run time equation. Once the modeller user gets past the hurdle of learning to program with components in C++ Builder, programming with the tool and its components can be fairly simple, and in true rapid application development perspective. A useful simulation model can be put together in a matter of few hours. Equation modelling in C++ Builder is entirely the programmer's responsibility as there are no component specifically tailored to this use. However, this is a straightforward process of writing equations in the code editor and compared to the diagramming process in simulators, saving on development time.

4.5 Application Cost of Approaches

There are two aspects that were considered in evaluating costs involved in the implementation of each approach, the cost of the software and the model development cost in developing a pilot model. Procuring the software for each approach depends on whether its application is for commercial or educational use. Taking the educational use as an example, it costs about three times more to purchase the simulator software than it does to purchase C++ Builder except in case of Vensim, which is free for educational use. In the other aspect, project cost will depend on development time, as CBP enables rapid application development, model development time with C++ Builder can be said to be comparable to that of a simulator, based on our limited experiments to date.

5.0 Summary and Conclusion

The overall performance and capability of each approach can be evaluated from the model produced from each environment. The advantage of dedicated simulation packages is in their effectiveness, the ease of simulating a complete or part complete model, and the accuracy obtained from a tool that has been specifically developed and tested for this purpose. The main argument, against it therefore is the limitations, of functionality and flexibility which is an unavoidable part of a packaged solution such as these "simulators".

The model interface which is easy to understand and use and it provides easy access to model equations and properties. It also provides an interesting and entertaining simulation environment, which enhances client interaction. However, the behaviour of simulation output from the programmed model is currently not satisfactory and as with any software development project its accuracy needs to assessed over a period of time and modifications made to achieve a level of robustness which would make it reliable to use.

This paper has described the application of CBP and a Simulator to the development of an experimental SD simulation model. Based on this pilot study, it is possible to highlight the potential of CBP the capability and flexibility of the SD approach. It was possible to create an interesting simulation environment within a short period of time,

but it does not fully reflect the extent to which components can change the way in which SD models are created.

This was due to the current lack of pre-fabricated components tailored specifically to this use. In view of this, we propose further work to implement a method for modifying existing components for use as simulation components.

It is possible to develop both simple and fairly complex applications with components doing very little programming with the use of edit boxes and some other components. Programming a SD simulation model requires a greater level of technical skills and, in the absence of suitable simulation components, the benefits of component-based programming cannot be successfully applied to SD simulation modelling. The level of programming required in the use of components depends on the degree of functionality built into them. It is therefore possible to develop SD simulation components, which would require little or no programming skills.

Acknowledgements

The authors would like to acknowledge the contribution of Angela O'Connor in conducting the simulation experiments on which this paper is based.

References

Umar, Imrana., (1997) A process for design and modelling with Components,. In Eds. Y. Balas, Vedat and Dicker 15th International Systems Dynamics Conference, Istanbul, Turkey pp 331-334

Balci, O., A. I. Bertelrud, C. M. Esterbrook, and R. E. Nance. 1997. Developing a library of reusable model components by using the visual simulation environ-ment. In *Proceedings of the 1997 Summer Computer Simulation Conference*, 253-258. SCS, San Diego, CA.

Balci, O., A. I. Bertelrud, C. M. Esterbrook, and R. E. Nance. 1998. Visual simulation environment. In *Pro-ceedings of the 1998 Winter Simulation Conference*, ed. D. J. Medeiros, E. Watson, J. Carson, and M. S.Manivannan. IEEE, Piscataway, NJ.

Borland. (1997). C++ Builder for Windows 95 & Windows NT: Component Writer's Guide. Borland International.

Brown, A. W., Ed. 1996. *Component-Based Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA.

Coyle, R. G., (1996) System Dynamics Modelling : A Practical Approach, Chapman and Hall., London

Harrell, C. and D. Hicks (1998) Simulation software component architecture for Simulation-based enterprise applications, *Proceedings of the 1998 Winter Simulation ConferenceD.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.*, 1717-1721.

Law, A.M., and Kelton, W.D. (1991). Simulation Modelling & Analysis. McGraw-Hill.

Mackulak, Gerald T., Frederick P. Lawrence., Theron Colvin (1998) An Effective Simulation Model Reuse: A Case Study for AMHS Modelling In *Proceedings of the 1998 Winter Simulation Conference Eds: D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, 13-16 December, Washington, DC 979-984*

Matko, D., Karba, R., Zupancic, B. (1992). Simulation and Modelling of Continuous Systems: A Case Study Approach. Prentice Hall International.

Miller, John A.Yongfu Ge, Junxiu Tao., Component-Based Simulation Environments: JSIM As A case Study using Java Beans. In *Proceedings of the 1998 Winter Simulation Conference Eds: D.J.*

Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, 13-16 December, Washington, DC 979-984
Pidd, M. (1995). Computer Simulation in Management Science. Third Edition. John Wiley & Sons, Inc.
Schmidt, B. (1986). Classification of Simulation Software. Systems. Analysis. Model Simulation. 3, pp
Hill, D.R.C. (1996). Object-Oriented Analysis and Simulation Addison-Wesley Publishers Ltd.