COMPONENT-BASED TOOLS FOR EDUCATIONAL SIMULATIONS

A thesis submitted in partial fulfilment of the requirements for the

Degree

of Doctor of Philosophy in Computer Science and Software Engineering

in the University of Canterbury

by Ruzelan Khalid

University of Canterbury

2013

ABSTRACT

e-Learning is an effective medium for delivering knowledge and skills. In spite of improvements in electronic delivery technologies, e-Learning is still a long way away from offering anything close to efficient and effective learning environments. To improve e-Learning experiences, much literature supports simulation based e-Learning. This thesis begins identifying various types of simulation models and their features that induce experiential learning. We focus on designing and constructing an easy-to-use Discrete Event Simulation (DES) tool for building engaging and informative interactive DES models that allow learners to control the models' *parameters* and *visualizations* through runtime interactions. DES has long been used to support analysis and design of complex systems but its potential to enhance learning has not yet been fully utilized. We first present an application framework and its resulting classes for better structuring DES models. However, importing relevant classes, establishing relationships between their objects and representing lifecycles of various types of active objects in a language that does not support *concurrency* demand a significant cognitive workload. To improve this situation, we utilize two design patterns to ease model structuring and logic representation (both in time and space) through a *drag and drop* component approach. The patterns are the *Delegation Event Model*, used for linking between components and delegating tasks of executing and updating active objects' lifecycles, and the *MVC (Model-View-Controller)* pattern, used for connecting the components to their graphical instrumentations and GUIs. Components implementing both design patterns support the process-oriented approach, can easily be tailored to store model states and visualizations, and can be extended to design higher level models through hierarchical simulation development. Evaluating this approach with both teachers and learners using *ActionScript* as an implementation language in the Flash environment shows that the resulting components not only help model designers with few programming skills to construct DES models, but they also allow learners to conduct various experiments through interactive GUIs and observe the impact of changes to model behaviour through a range of engaging visualizations. Such interactions can motivate learners and make their learning an enjoyable experience.

ACKNOWLEDGMENTS

I wish to sincerely thank my supervisor, Associate Professor Dr. Wolfgang Kreutzer and my associate supervisor, Professor Dr. Tim Bell for all their constant intellectual challenges and very kind guidance and encouragement during this study.

I would also like to thank all staff and postgraduate students at University of Canterbury for whatever help they gave to complete this study.

To my family, thanks so much for giving your continuous moral support and encouragement, and sharing your valuable time during our stay in New Zealand. You all have always been my source of strength and inspiration.

Lastly, thanks to all of those who implicitly or explicitly committed until the completion of this study.

TABLE OF CONTENTS

ABSTRACT i

ACKNOWLEDGEMENTS ii

LIST OF FIGURES vii

LIST OF TABLES ix

1. INTRODUCTION 1

1.1 Introduction 1

1.2 Statement of the Problem 4

1.3 Objectives and Motivations 4

1.4 Scope of the Research 12

1.5 Contributions to Knowledge 14

1.6 Thesis Overview 17

2. SIMULATION AND EDUCATION 19

2.1. Introduction 19

2.2. Simulation Models and Their Purposes 20

2.3. Types of Simulation Models 22

2.4. The Role of Simulation in Education and Learning 26

2.4.1 The Role of Simulation in Learning Theories 26

2.4.2 Empirical Evidence 32

2.4.3 Simulation and e-Learning 33

2.4.3.1 Promises and Problems of e-Learning 33

2.4.3.2 The Roles of Course Management Systems 35

2.4.3.3 Pedagogical Aspects of e-Learning 36

2.5. DES Development Tools 38

2.6. Animated DES Systems 40

2.7. Summary 44

3. **A FRAMEWORK FOR DES AND ANIMATION** 46
3.1. Introduction 46
3.2. DES and Queuing Scenarios 48
3.3. Modelling Time 50
3.2.1 The Event-Oriented Approach 51
3.2.2 The Process-Oriented Approach 52
3.4. The DES Framework 54
3.4.1 The *Data Collector* Package 56
3.4.2 The *Distribution* Package 57
3.4.3 The *Monitor* (Simulation Executive) Package 59
3.4.4 The *Resource* (Servers and Queues) Package 61
3.5. Graphical Objects in Discrete Event Models 62

4. USING FLASH FOR SIMULATION 67
4.1 Introduction 67
4.2 Visual Simulation and Visual Interactive Simulation 68
4.3 Animation Approaches 71
4.4 Managing Simulation and Animation 74
4.5 Flash as an Implementation Language for Simulation and Animation 77
4.5.1 Flash Features for VIS Development 78
4.5.2 Flash Component Construction 79
4.5.3 Other Advantages of Flash and Its Drawbacks 81
4.6 Flash Components for Queuing Systems 83
4.7 Flash Components for Visualizing Queuing Systems 89
4.8 Example 91
4.9 Problems and Pitfalls 96
4.10 Extensibility 99

5. **COMPONENT-BASED MODELING FOR ANIMATED SIMULATION** 102
5.1 Introduction 102
5.2 Component Based Simulation 104
5.3 The Environment of Animated Simulation Models 105
5.4 The Delegation Event Model for Linking Components 107
5.5 The MVC for Visualizing Component States 111

5.6 Connecting External Data 114
5.7 Example 118
5.8 Towards Hierarchical Simulation Model Designs 122
5.9 Designing Mechanisms for Hierarchical DES Models 125
5.9.1 Monitor Delegation Mechanism 126
5.9.2 Monitor Communication Mechanism 130
5.10 Problems and Challenges 133

6. **EVALUATION AND ANALYSIS** 136
6.1 Introduction 136
6.2 Evaluating Models' Attractiveness and Interactivity 137
6.2.1 Assessment and Evaluation Methods 137
6.2.2 Experiment Participants 139
6.2.3 Data Analysis and Results 142
6.2.3.1 General Information 142
6.2.3.2 General Questions 143
6.2.3.3 Model Rating 145
6.3 Evaluating the Tool's Ease of Use, Usefulness and Enjoyment 159
6.3.1 Assessment and Evaluation Methods 159
6.3.2 Experiment Participants 160
6.3.3 Running the Experiment 162
6.3.4 Data Analysis and Results 164
6.3.4.1 General Information 164
6.3.4.2 Questionnaire Reliability and Validity 165
6.3.4.3 Usefulness, Ease of Use and Enjoyment of the Tool 166
6.3.4.4 Self Predicted Future Usage 168
6.3.4.5 Participants' Cognitive Workload 171

7. **CONCLUSION AND FUTURE RESEARCH** 175
7.1 Introduction 175
7.2 Conclusion 175
7.3 Limitations of the Research 179
7.4 Recommendations for Future Research 181

BIBLIOGRAPHY 173

APPENDICES

Appendix A: Consent Form

Appendix B: Questionnaire Information Sheet

Appendix C: Learner Questionnaire

Appendix D: Model Builder Questionnaire

Appendix E: User Manual

Appendix F: Source Code (in CD)

LIST OF FIGURES

Figure 1.1	Interactions between Teachers, Learners, Models and LMSs	11
Figure 3.1	The Event-Oriented Approach Mechanism	52
Figure 3.2	The Process-Oriented Approach Mechanism	53
Figure 3.3	Package Diagram for Queuing Models	56
Figure 3.4	Class Diagram for the *DataCollectors* Package	56
Figure 3.5	Class Diagram for the *Distribution* Package	58
Figure 3.6	Class Diagram for the *Monitor* Package	59
Figure 3.7	Class Diagram for the *Resource* Package	61
Figure 3.8	*Graphical Objects* in DES	63
Figure 4.1	Visual Simulation Components	69
Figure 4.2	Three Approaches to Combine Simulation with Animation	71
Figure 4.3	DES's Animated Objects	75
Figure 4.4	Transformation from *Model* to *Animation* Time	76
Figure 4.5	Component Architecture	79
Figure 4.6	Class Diagram of *Components* for Simulation *Input and Output*	84
Figure 4.7	Flash Component Panel	87
Figure 4.8	Samples of DES Visualization Tools	91
Figure 4.9	Sample of Interactions between Learners and a Model	95
Figure 4.10	Sample of Information Gained from a Model	96
Figure 4.11	Extended Components for Supporting Logistic and Manufacturing Systems	99
Figure 5.1	Simulation and Animation Aspects of a Model	105
Figure 5.2	The DES Delegation Event Model Structure	108
Figure 5.3	The flow of a SimProcess Object in DES Components	110
Figure 5.4	The DES *MVC* Structure	112
Figure 5.5	Flash Development Environment	118
Figure 5.6	A Queuing Network System	119
Figure 5.7	A Server's Properties and Default Values	120
Figure 5.8	A Final Model	121
Figure 5.9	Interactions with Component Instances	122
Figure 5.10	Hierarchical Construction of a DES Model	124
Figure 5.11	Submodel Architecture and Transferring Mechanisms	126

Figure 5.12 Monitor Delegation Mechanism 128
Figure 5.13 Submodel Class Definition 129
Figure 5.14 Simulation Class Definition 129
Figure 5.15 Agenda States 132
Figure 6.1 Simple Queuing Networks 141
Figure 6.2 More Complicated Queuing Networks 141
Figure 6.3 Participants' Feedback on Simulation Knowledge 144
Figure 6.4 Arena Screenshot 161
Figure 6.5 Perceived Usefulness Results 167

LIST OF TABLES

Table 2.1	Classification of Constructive Computer Simulations	23
Table 2.2	Simulation Types and Learning Support	24
Table 2.3	Some Learning Theories and Their Features	28
Table 2.4	Available DES Simulation Tools	38
Table 2.5	Desirable Features for the Design of DES Tools	44
Table 3.1	Types of Directed Graphs	64
Table 3.2	Properties and Events for Dynamic Objects	65
Table 4.1	Aspects of Simulation-Animation Approaches	73
Table 4.2	Interaction Characteristics of Concurrent and Post-processed Animations	74
Table 4.3	Available Simulation Tools and Their Features	74
Table 4.4	Simulation to Animation Conversion	75
Table 4.5	Events and Model Time Difference in a Sample System	76
Table 4.6	VIS Graphic Displays and Flash Features	78
Table 4.7	DES Component Types	86
Table 4.8	Flash Components for Building DES Models and Their Functionalities	86
Table 4.9	Flash Components for Visualizing DES Models and Their Functionalities	90
Table 5.1	Server Properties and Description	120
Table 6.1	Items in Model Rating	140
Table 6.2	Time Spent (in minutes) for Each Score	144
Table 6.3	Good Simulation Knowledge Participants' Feedback about the Models	146
Table 6.4	No Simulation Knowledge Participants' Feedback about the Models	146
Table 6.5	Undecided Simulation Knowledge Participants' Feedback about the Models	147
Table 6.6	Feedback on the Quality of Animation from the Participants Who Always Used Computer as a Learning Tool	149
Table 6.7	Sub-questions of "These tools help to understand the model better (Please write if you have any comments)"	153
Table 6.8	Good Simulation Knowledge Participants' Feedback about the Model Tools	153

Table 6.9	No Simulation Knowledge Participants' Feedback about the Model Tools	154
Table 6.10	Undecided Simulation Knowledge Participants' Feedback about the Model Tools	154
Table 6.11	TAM Factors and Their Variables	160
Table 6.12	Items of Perceived Ease of Use, Perceived Usefulness, Perceived Enjoyment and Self-predicted Future Usage of the Component-based Tool	163
Table 6.13	The Participants' Gender	164
Table 6.14	The Participants' Knowledge and Experiences	164
Table 6.15	Cronbach's Alpha Values	165
Table 6.16	Factor Analysis of Perceived Usefulness, Perceived Ease of Use and Perceived Enjoyment	166
Table 6.17	Descriptive Statistics of the Items	167
Table 6.18	Descriptive Statistics of Self-Predicted Future Usage	168
Table 6.19	Correlations between Perceived Usefulness, Perceived Ease of Use and Perceived Enjoyment to Self-Predicted Future Usage	169
Table 6.20	Regression Analyses of the Effect of Perceived Usefulness and Perceived Ease of Use on Self-Predicted Future Usage	170
Table 6.21	Participants' Feedback about the TLX Subscales	172

CHAPTER 1

INTRODUCTION

1.1 Introduction

e-Learning (i.e., technologies that use digital technologies to deliver and facilitate learning) is increasingly used in schools, higher education and training centres either to support distance learning or to complement the traditional classroom environment. Since it uses electronic media; e.g., the Internet, to support learning, this style of knowledge transmission eases traditional constraints on time, space and distance. The advantage to learners is that they can learn at anytime and anywhere. As a result, the use of e-Learning has grown rapidly throughout the world. However, this technology requires that learners themselves are responsible for gaining knowledge; a key concept of *learner-centred* education.

The teacher-student ratios either for primary, secondary or tertiary education in some countries (e.g., India, South Africa, Philippines, etc.) are still high. In India, the teacher-student ratio for secondary school was reported 32.7 in 2004 and 25.33 in 2010 (http://www.tradingeconomics.com). Although the ratios have slightly been improved in most countries during past few years, less time dedicated by teachers to the needs of each individual student demands attractive and interactive learning materials to promote and enhance their learning experiences. Learning materials that focus on *activities* (i.e., some degree of interaction) during the learning process are crucial in this and have proved to have more positive impacts on learning than static materials, such as numbers, texts and pictures (Holzinger & Ebner, 2003; Neumann, Page, Kreutzer, Kiesel, & Meyer, 2005; L. P. Rieber, 1996). Multimedia materials that allow content navigation that integrate texts, pictures, diagrams, sound and dynamic images (i.e., animations and movies) are increasingly integrated in learning environments. More recently, techniques that make learning more enjoyable and fun

(e.g., simulations and computer games) have also been proposed (e.g., see Aldrich, 2002, 2004, 2005; Prensky, 2001).

Simulation is a technique for experimenting with models of real or imaginary systems (see Aldrich, 2002, 2004, 2005; Prensky, 2001). Since it allows learners to manipulate parameters and directly observe the impact of modifications on model behaviour and performances, it can be a powerful learning tool, whose "hands-on" activities engage learners emotionally and help to improve understanding of complex scenarios. There is a large body of literature (e.g., C. N. Quinn, 2005; Rosson & Seals, 2001; Smialek, 2002; Syrjakow, Berdux, & Szczerbicka, 2000; Thomas & Milligan, 2004) that corroborates these benefits of simulations in a learning and teaching environment.

The main benefit of embedding simulations in an educational context is that it stimulates a scientific *discovery* style of learning; i.e., learning based on self-directed initiatives (Jong & Joolingen, 1998; Neumann et al., 2005; L. P. Rieber, 2002). This learning style requires learners to initiate and control their knowledge acquisition through designing and executing experiments, analyzing model feedback and constructing hypotheses based on this information (River & Vockell, 1987). The iterative cycle of experimentation and drawing conclusions from exploring a model are believed to encourage critical thinking, scaffold a deeper and more structured understanding of concepts, and encourage long lasting retention of a learned domain (Aldrich, 2004, 2005; Schwartz, Bransford, & Sears, 2005).

In spite of its strengths, simulation-based learning is an unsupervised learning environment whose effectiveness depends strongly on learners' and models' characteristics, and how much guidance can be provided. Learners' characteristics include learning styles (Martinez, 2000), motivation (Wittrock, 1989), prior knowledge (Dochy, Segers, & Buehl, 1999; Hailikari, Katajavuori, & Lindblom-Ylanne, 2008), meta-cognitive aspects (i.e., strategies for directing learning) and other miscellaneous skills (Joolingen & Jong, 1991b; Njoo & Jong, 1993; White, Shimoda, & Frederiksen, 1999). Among these factors, prior knowledge of a studied domain tends to have the strongest influence on effective exploration (Lee, 1999; Mayer, 2003). Without such knowledge, learners tend to suffer ineffective and inadequate exploration. Ineffective exploration leads learners to insignificant experimentations and difficulties in drawing conclusions from model experiments, while inadequate exploration tends to result in too shallow understanding. Thus, some researchers

(e.g., J. R. Anderson, Corbett, Koedinger, & Pelletier, 1995; Kirschner, Sweller, & Clark, 2006) urge teachers to support learners with guidance that directs learning and aids their activities. Examples of suggested guidance are structuring tasks through explicit instructions (Veermans, Jong, & Joolingen, 2000), requesting learners to observe and describe interesting scenarios (Tan & Biswas, 2007), or guiding learners at appropriate times; e.g., through *Adaptive Coaching for Exploration* (Bedor, Mohamed, & Shedeed, 2004; Bunt, Conati, Huggett, & Muldner, 2001; Bunt, Conati, & Muldner, 2004; Noguez & Sucar, 2005).

While guidance is important for directing learning, models should act as platforms for testing hypotheses. Experimentation and deduction is only possible if models contain these features:

- *activities* (e.g., mouse clicking/rolling, keyboard input, etc.) to motivate learners' actions and challenge their imagination,
- informative and meaningful *feedback* and *visualizations* (e.g., through texts, images, diagrams, graphs, sounds, etc.) that motivate learners to perform further experiments,
- attractive responsive *animations* that demonstrate feedback of model behaviour, and
- reflection of *real world scenarios* that stimulate learners' imagination and connect their mental models to the outside world.

To draw good conclusions, learners not only need to engage and interact with a model, but also need to communicate with their peers and teachers. Fortunately, facilities for this are widely available in modern *Learning Management Systems* (LMSs). To fully complement e-Learning environments, they need attractive, interactive and informative learning materials. Prior to 1996, the development of such materials was highly dominated by Java (Arnold, Gosling, & Holmes, 2006; Lambert & Osborne, 2004). Since then the development of highly interactive models has been made easier by the introduction of the Adobe's *Flash* animation tool (Castillo, Hancock, & Hess, 2004; Stenalt & Godsk, 2006). However, this multimedia

development tool has not been utilized to its full capabilities to support learning and teaching.

1.2 Statement of the Problem

In spite of improvements in electronic delivery technologies, e-Learning is still a long way away from offering anything close to efficient and effective learning environments. To be effective, electronic courseware management (e.g., through LMSs) requires high quality contents such as simulations and educational games to improve e-Learning experiences. Unfortunately, common LMSs offer no support for this and little research has been done to ease the construction and customization of online simulation models and their integration into learning management systems. As a result, e-Learning is still dominated by static materials (e.g., PDF, Microsoft Word and PowerPoint files, etc.), rather than more sophisticated and dynamic techniques; some detailed data is given in Wagner (2006).

While much has been claimed about the benefits of simulations and games in supporting and enhancing learning and training, few investigations into how to develop and construct simulation tools, how to design attractive and interactive model graphical user interfaces (GUIs), how to store models' intermediate states, and how to integrate simulations into LMSs have been performed. To improve this state of affairs, it seems important to make both model construction and model deployment easy for teachers, so that the resulting models are attractive and interactive enough to motivate learners to explore and experiment, and so that tools can easily be extended to help model developers to construct libraries for painless construction of many different types of animations and visualizations.

1.3 Objectives and Motivations

This research assumes that simulation models are useful tools for clarifying ideas and showing flows of events. It is therefore *not* our primary objective to demonstrate that simulations enhance student learning - an assumption that has already been corroborated by many empirical investigations (e.g., Gokhale, 1996; Liao & Miller,

1996; Michael, 2000; Renshaw & Taylor, 2000; L. P. Rieber, 1996). Instead, this research investigates how simulation models can most easily be built and delivered within an e-Learning environment. We focus on *Discrete Event Simulation* (DES) models. Thus, the research plans are to:

- design and construct a *tool* for animated simulation models for web based delivery and LMS integration
- integrate the models with suggested *model features* that facilitate learning
- analyse users' feedback of the tool and its resulting models
- extend the tool to support more complex models

Our motivation is clear. We found no tools that allow users to interact with their resulting models, customize the models' visualizations during runtime and save the models' states and animations at any point of interest for later uploading. Thus, our particular interests centre is on exploration, construction and application of DES tools that can effectively support three groups of users:

1. *developers* (i.e., those who are interested in extending these tools to new applications),
2. *teachers* (i.e., model designers and implementers) and
3. *learners* (i.e., model users).

Developers should be conversant with the tools' internal architecture, so that extension is easy and not unduly limited. Teachers, on the other hand, need easy-to-use model construction tools, since they are probably lacking in programming knowledge and experiences. Finally, learners should be presented with attractive and interactive animated models that support knowledge acquisition through experimentation.

To satisfy all three parties' expectations, a visual modelling environment that offers component-based composition of simulation models has been designed and constructed. It reduces model complexity through use of pre-assembled components,

each of which handles their specific functionality. These components can be combined to form models. This approach eases model construction since components can be reused over and over again. Component development is based on an Object Oriented architecture (Eden, 2002; Lau, 2000) and the design of their code follows Object Oriented Programming (OOP) principles of good practice with regard to encapsulation, inheritance, polymorphism and exception handling.

We identified two design patterns that suit the development and extension of the DES tool; i.e., the *Delegate Event Model* and the *Model-View-Controller (MVC) interface architecture*. The *Delegate Event Model* was used to wire components to each other, since its style of event broadcasting is analogue to the flow of entities in DES components, so that that an entity (an *event object*) is passed from a component (an *event source*) to other components (*event listeners*). The *Model-View-Controller (MVC) interface architecture* is used to support a component's graphical interfaces (GUIs) and multiple visualizations of its states. By following this design pattern, components can be loosely coupled to their GUIs (to receive inputs) and visualizations tools (to receive state notifications). Adding or removing visualizations does not affect other component parts since each component only store a list of interested visualization instances - without any influence on a visualization's implementation. Since each component needs to perform two tasks; i.e., communicating with each other and notifying state changes to an observer, the component's class must define both patterns in its implementation.

The component-based modelling framework offers ease-of-use by allowing model designers to drag components from a library, drop them onto a worksheet and assemble them appropriately into models. Four categories of simulation components have been designed and implemented:

- components for *modelling* activities,
- components for *visualizing* simulation results,
- a component for controlling *animation* speed, and
- a utility component for *saving or refreshing* model states and *revealing* their flows or lifecycles.

Various component properties can be customized through GUIs. Since *modelling components* have output port properties (i.e., they store a list of interested components that wish to receive state change notifications), they must be wired to each other so that messages can be routed in the right order. When all components have been wired together into a model, teachers can test and then distribute the model to learners. Although the resulting model has a fixed structure, we have tailored the components to allow learners to change model parameters and explore the resulting chains of events without any need to change model code. Since each component is also an object, the values for the output port properties can be specified during runtime.

We have identified five elements that should exist in a DES model to help learners understand its behavior; i.e.:

1. A model should provide easy-to-access runtime GUIs for changing component parameters. These could employ mouse-over to allow learners to quickly view a component's attribute values, text boxes to receive input-based interactions (e.g., time of an entity's creation, a resource's capacity, etc.), combo boxes to permit learners to type a value directly into a field or choose a value from a list of existing options (e.g., queuing disciplines, distributions that specify time between arrivals, delays, resources' service times, etc.) and command buttons to activate visualization tools (e.g., graphs, histograms, box plots, etc.). Data visualization tools should be easy to be added, removed, sized and positioned at any location through drag and drop gestures. To make their display both more informative and attractive, some model components; e.g., servers, should be animated to depict their current states.
2. A model should offer a display list of all past, current and next events, so that learners can obtain clarification on how it is executed and how component parameters affect event sequences in the model. Without such a list, learners tend to just passively view animations rather than actively seeking an understanding of model behaviour; i.e., how events are affected by different model parameters.
3. A model should animate message passing and movements of transient entities between components. Arrows can depict a message's or an entity's travel direction, but learners should be able to remove this feature if it obscures other patterns or visualizations.

4. A model should provide a high degree of top-level control over a simulation and its animation; e.g., allowing learners to stop, restart, speed up or slow down the execution of models and their animations. This gives learners a choice to look closer at aspects that catch their attention and skip over aspects that are of no current interest. While such a capability is helpful in fostering understanding, proper synchronization of animation speed and simulation clock time is crucial to preserve a consistent correspondence of simulation and animation activities.
5. A model should provide a utility component for allowing learners to *save* model visualizations and entities' current states for restarts or reloads of a model without the need to exit from the program or refresh a web page.

Embedding these functionalities in a model however poses a number of challenges. These include:

1. The construction of runtime GUIs is only possible through an Application Programming Interface (API). Since component GUIs are based on the *MVC* pattern, this demands that each component must be equipped with its own GUI to handle its parameters. When there are many components, this is a cumbersome task.
2. While there could be many attractive and interactive third-party data visualization components on the market, they cannot be easily integrated with our components. The main reason once again lies in the implementation of the *MVC* pattern, which demands that all interested observers (i.e., visualization tools) define an *update* method in order to receive notifications from the components. We have therefore opted to implement our own data visualization constructs.
3. Implementing the *Delegation Event Model* pattern in an animated simulator requires to correctly trigger sorted events in the *Monitor* at appropriate times (i.e., to stop or delay events appropriately before attempting to trigger subsequent events) and to smoothly transfer entities along their life cycles so that they reach their next destination at times that are consistent with the *viewing ratio* (i.e., animation speed) specified by a learner. This necessitated a nested design, where model time must be mapped onto animation time, and animation time then mapped onto real time. We have therefore opted for *concurrent animations* to

immediately display the effect of viewing ratio changes, rather than a *post-processed animations* or *direct simulation-animation* (Hill, 1996) architecture.

4. Storing models requires storing all component instance identities (with their current states and all interested observers) and running the models requires continuation from their last saved positions (e.g., entities must continue travelling to their next location based on their current locations and leftover travel times). We therefore investigated methods to perform these.
5. Since we also designed our components to support *hierarchical* simulations that can accommodate more complex model structures, we need to find a way to connect and synchronize models in a hierarchical fashion, where aspects of parent models may depend on their child model(s) states. This demands a mechanism that not only synchronizes the flow of simulation entities in a child model, but can also transmit this information to its parent whenever its relevant events have been executed.

Before providing such components, we had to construct core libraries for coordinating state transitions and processes in DES models; i.e., a DES *monitor* engine. Its purpose is to keep track of all DES aspects, such as entities, resources, routing, buffering, scheduling, time management and statistical instrumentation. To achieve this goal, it had to be possible to generate samples from a variety of distributions, maintain a list of events to be executed, offer a mechanism for generating and cancelling events, maintain a simulation clock, compute statistical performance measures (e.g., minima, maxima and averages of time spent in a system, waiting times in queues, resource utilization, throughput, etc.) and collect and display the results of a simulation run.

Since these models are intended to be embedded in web pages and meant to drive animations, we have used Adobe's *Flash* (Lopez, 2006; Peters & Yard, 2004; Sanders, 2004) for coding their implementation. Flash was chosen as a delivery platform mainly because of its strength as an animation tool (Holzinger & Ebner, 2003; Mohler, 2006; Peters & Yard, 2004; Shupe & Hoekman, 2006), and the fact that it can generate very compact *.swf* applets that can be played "off the shelf" in the vast majority of modern browsers.

Although there are a number of Java-based simulators; e.g., simjava (W. Kreutzer, J. Hopkins, & M. V. Mierlo, 1997), JSIM (J. A. Miller, Y. Ge, & J. Tao, 1998), JavaSim (Kuljis & Paul, 2000; Tyan, 2002), Psim-J (Garrido, 2001) and Desmo-J (B. Page & Kreutzer, 2005) and some simple device modelling tools for operating cell phones, crane arms, etc. using Flash (e.g., Kaye & Castillo, 2003), we have not found any reports or references to a Flash-based discrete event modelling tool. We have therefore coded our own Flash-based DES model executive. This meant that we first needed to learn how to use Flash's development environment, its object-oriented scripting language (ActionScript-2), both its generic and animation specific libraries, and its features for building and packaging collections of reusable components. Although the construction of such a DES engine was not a primarily goal of this research, its development has been a necessary step in providing a suitable *infrastructure* for subsequent work. Learning how to build such a DES monitor in ActionScript and how to package it so that its features can be easily used, took a significant amount of time.

After coding the basic libraries, we fine-tuned our components so that they could support all aspects and model features we have mentioned above. To test their effectiveness, two experiments were conducted. First, we obtained feedback from learners about the attractiveness, interactivity and usefulness of our Flash components in the context of two DES sample models. Secondly, we distributed the components to model developers to get their feedback about the tool's usefulness, ease of use and enjoyment. Here the information collected included whether the components provided interesting run time GUIs, whether the GUIs were easy to interact with, whether the learners liked the approach to display visualizations only when requested, which visualization tools (e.g., graphs, histograms, boxplots etc.) helped them to understand models better, whether the ability to change simulation parameters during run time and the ability to pause, slow down and speed up a model's execution made learning easier and/or more enjoyable, etc.

The resulting models should easily be embedded in LMSs. Fortunately, Flash models can easily be tailored to handle communications between learners and LMSs compared to the use of JavaScript in HTML files as in the traditional approach. The main justification for the integration was to take advantages of LMS facilities such as:

- collecting information of learners' behaviour,
- allowing access to online forums or chat rooms that increase collaboration between learners, or learners and teachers, and
- improving integration with other learning materials.

Additional reasons are to present learners with a uniform interface (thereby minimising any distractions from focussing on what they are meant to learn) and to ease model maintenance, so that models can regularly be updated without any need to distribute new copies to all learners. Figure 1.1 shows a sketch of the interactions between teachers, learners, simulation models and a LMS. Their interactions can briefly be described as follows. Teachers translate their mental models to computer models using the right tools. The computer models are then distributed into a LMS where they can be viewed by remote learners. Learners interact with the models and the feedback from such interactions will automatically be displayed to them. If they need further clarification on the feedback, they can use the LMS's facilities (e.g., chat rooms, email, etc.) to interact with their teachers or peers.

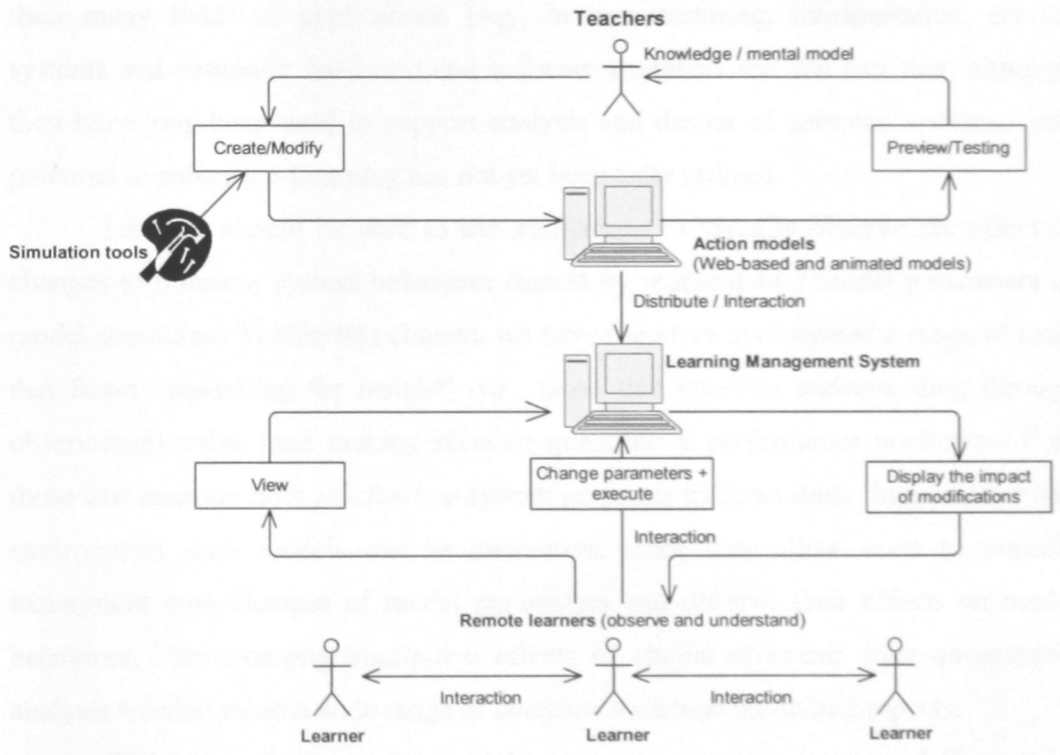


Figure 1.1 Interactions between Teachers, Learners, Models and LMSs

1.4 Scope of the Research

There are two types of simulation models: static and dynamic. In a *static* model, time does not affect model behaviour. Examples are device simulations (Kaye & Castillo, 2003), equipment operation simulations (Towne, 2007) and so called "soft skill simulations" (Aldrich, 2005; Gaffney, Dagger, & Wade, 2008; Maldonado et al., 2005; Vries, 2004). These kinds of simulations are familiar to teachers and their use in educational environments has long been discussed (e.g., see Aldrich, 2004, 2005; Gibson, Aldrich, & Prensky, 2007). *Dynamic* models trace behaviour that changes over time. Examples are DES, where system behaviour spawns a sequence of discrete events, and system dynamics models, where the system behaviour is described through sets of equations that model how states fluctuate "quasi-continuously" over time.

This research has concentrated on DES models, where the state of a model changes only at specified points in time, and more specifically on *Queuing Networks,* which explore the effects of capacity constrained resources on common performance measures; such as response time and throughput. This choice was made because of their many fields of applications (e.g., in manufacturing, transportation, service systems and computer hardware and software analyses) and the fact that, although they have long been used to support analysis and design of complex systems, their potential to enhance e-Learning has not yet been fully utilized.

Learners should be able to use animations to visually observe the effect of changes to transient system behaviour caused by manipulating model parameters or model structures. Within this context, we have therefore investigated a range of tools that foster "modelling for insight" (i.e., those that improve understanding through observation) rather than making accurate quantitative performance predictions (i.e., those that measure how *efficiently* a system performs its functions). In an e-Learning environment such models can be instructive, since they allow users to visually experiment with changes of model parameters and observe their effects on model behaviour. By stressing *qualitative* effects of chains of events over *quantitative* analysis we also avoid a wide range of complex statistical modelling aspects.

Within the discrete event modelling domain, two dominant modelling styles (world views) are typically used to control flows of events: event-orientation and

process-orientation. While *event orientation* eases coding, *process orientation* offers a more natural framework for model development. Our designs therefore use a process-oriented approach. Unfortunately this causes some implementation issues (e.g., the lack of a built-in *coroutine* or continuation features in most common implementation languages).

Since DES has the ability to model complex systems with relative ease, many commercial or research tools have been developed for constructing DES models. However, these tools are typically targeted at *analysis* rather than learning purposes. Many commercial simulation software; e.g., Arena (Kelton, Sadowski, & Sturrock, 2004), Flexim (Nordgren, 2003) and SIMUL8 (Concannon, Elder, Tremble, & Tse, 2006), are excellent tools for building sophisticated simulation models and observing model behaviour through animation. However, the resulting models mostly lack support for user-experimentation during run time, are operating system dependent, must be run using a specialized proprietary software, and are not designed to execute on a web page; a very important element for incorporating models in e-Learning systems. Thus, investigations on how these constraints can be catered are crucial.

In order to support web-based models, most previous research tools in this domain have been developed in Java. Two web-based approaches can be distinguished: *Web-supported simulation* and *Web-enabled simulation*.

Web-supported simulation locates tools on a server that can then be accessed to create and run models. Thus, users do not have to install software packages on their machines. Examples include JSIM (J. A. Miller et al., 1998), Silk (Healy & Kilgore, 1998; Kilgore, 2000), JavaGPSS (Kazymyr & Demshevska, 2001; Klein, Straßburger, & Beikirch, 1998), WSE (Iazeolla & Ambrogio, 1998) and ASimJava (Sikora & Niewiadomska-Szynkiewicz, 2007). JSIM and Silk ease model constructions using component-based technology with Java Beans. However, among these tools, only JSIM integrates a simple animation for displaying queues.

Web-enabled simulation requires the installation of software packages on users' machines. Examples are Psim-J (Garrido, 2001), SSJ (L'Ecuyer, Meliani, & Vaucher, 2002), JavaSim (Tyan, 2002) and DESMO-J (Meyer, Page, Kreutzer, Knaak, & Lechler, 2005a). However, these packages, while giving experienced programmers the flexibility to code their own extensions, typically only support textual description and very simple data visualizations.

We chose the second approach. The main reasons are that we believe the first approach would be a burden on servers, since all development processes (e.g., model construction, execution and animation) must all be performed on a central server, and also limit tool accessibility, since it depends on network availability, its speed and the number of concurrent users accessing the servers.

1.5 Contributions to Knowledge

This research has made some positive contributions to simulations in education especially in proposing a design of DES tools for engaging and helping learners to understand DES behaviour. The design focused on methods of easing the construction of *attractive, interactive and informative web-based* simulation models. These contributions have been achieved through a various processes of investigating, analyzing and structuring how a DES tool can be provided with the right design.

In proposing the tool, we first surveyed the current use of simulation models in the learning and teaching environment. We then identified and made a critical analysis of model features that support *learner-centred learning* based on learning theories and previous literature review. This deserves to be investigated since educationalists and tool developers are considerably separated in their own domains. Educationalists keep proposing and proving the benefits of using simulations as a tool for learning and teaching in the new era of education, and how these benefits can be gained using the right models. The tool developers meanwhile concentrate more on the development of modelling and complete system analysis tools for measuring system performances. Thus, they typically ignore the educationalists' views of the right models that stress on the importance of interactions between learners and the models in ensuring learning. We are trying to bring both parties closer. Thus, we made an analysis of how simulation models could be better supported in the current learning and teaching environment by investigating and analysing the available DES software and packages to discover what tools and functions they provide and lack in facilitating learning and teaching. This can be a reference for those who intend to provide such the right tool.

The contribution that directly relates to the tool design was the proposal of strategies to construct and incorporate the tool with the suggested model features that

relieve learners' cognitive processes during their learning; i.e., *hypothesis test platforms*, *concurrent responsive animation* and *customized visualizations*. Before this work, no tools have been designed and constructed to support all the three features during model runtime. Moreover, we designed the tool so that its resulting models support a high degree of simulation and animation control and provide a store capability of their states, animations and visualizations at any simulation time points for future use. For this, we architected DES frameworks, extended them to various components (i.e., simulation building blocks) with well-defined interfaces and contracts that describe the input and output of entities and data flows, designed and tested the components, and recommended the use of appropriate design patterns for facilitating their constructions. To prove this design works, we managed to develop a proof of concepts of a DES tool. We believe that its use eases the constructions of attractive, interactive and informative DES models for self learning purposes.

Our design focused on the integrations of simulation, animation and visualization to reflect *change* in the time (i.e., when simulation encounters delays), space dimension (i.e., when an entity moves) and model states (i.e., when an event is executed). In an animated simulation environment, the time requires model time to be mapped onto animation time and animation time to be mapped onto real time, the space dimension requires a stage for constructing and locating animated entities and model structures, while model states require visualization tools (e.g., graphs, histograms, etc.) to display their abstract data. Investigating what elements should exist to fulfil these requirements and how they were supported by Adobe Flash, and arguing how best the Adobe Flash as a platform for the DES tool development were another research contributions.

To reflect users' feedback on our tool and to obtain their recommendations for its future improvement, we conducted two experiments. Conducting these experiments yielded two contributions. The first contribution was the analyses of learners' feedback about how significant relevant features (e.g., animations, visualizations, interactions, customized interfaces, etc.) of DES models helps them engage with and get insight into the models' behaviour. The analyses enabled us to compare and judge how consistent their feedback was with the previous claims that stress the importance of providing the features to ensure learning. The second contribution was the analyses of model designers' feedback about how good our tool

is in constructing DES models. The analyses enabled us to evaluate how useful and easy our tool is for constructing the DES models for learning and teaching purposes.

In addition to addressing the tool design for a single layer of DES models, we also architected how the design could be extended to manage the complexity of large and complex DES models. This complexity can either relate to the *cognitive* aspect (i.e., how model logics can be split to smaller models for representing their functions) or the *representational* aspect (i.e., how many elements are used and how they are arranged to represent model structures). Our approach of catering the complexity is through a *hierarchical structured concept*; i.e., by breaking up a model to relevant sub-models with each sub-model conceals the details of their lower levels. The concept manages both aspects through its ability in controlling the level of details (in terms of structures and information) for better representing of the model and arranging animation and visualization for better viewing and grasping the dynamic parts of the model (as opposed to the crowdedness of graphical objects in a flat model).

However, the main challenge for the design is the synchronization of each sub-model's behaviour so that they can be executed in the right order. For this, we present two mechanisms for coordinating event executions among layers in hierarchical DES models. These are the *Monitor Delegation Mechanism* that delegates event executions to a relevant layer and the *Monitor Communication Mechanism* that transfers event executions to all visited layers.

Our approaches differ from the approach proposed by Yi and Cho (Yi & Cho, 2001, 2003). We focused on how to extend our simulation engine and components based on the *concurrent animations* where a simulation monitor controls both simulation and animation aspects to guarantee animation accuracy. Since our components allow interactions, the runtime interactions with all layers are automatically supported. Their approach meanwhile is based on the *direct-simulation animation* where the simulator and the animator have their own activity scheduling lists. Thus, besides considering event executions among layers in the simulator, they also need to find a method of communicating the simulator with animation scheduling in the animator. The main drawback of their approach is that it only guarantees animation accuracy from event to event, not between them since the graphics rendering depends on the computer that simulator and animator reside.

1.6 Thesis Overview

This thesis is organised as follows. In *Chapter 2*, we first identify different types of simulation models, examine their roles in facilitating learning based on learning theories and collect some empirical evidence that establishes their effectiveness in e-Learning environments. Some available simulation tools and the current interests on animated DES models are also reviewed.

Chapter 3 examines the mechanisms of two DES modelling styles: event-oriented and process-oriented. Realising the limitations of the event-oriented style, we architected a process-oriented DES framework to support various DES facilities (e.g., generating random numbers for various distributions, collecting statistics, managing simulation clocks, a list of future events, etc.). This framework has guided the construction of our DES simulation libraries. To symbolize the libraries' functionalities and ease the building of DES models through *symbol compositions*, we then introduce relevant DES graphical objects. However, it still demands programming effort and its resulting models offer no support for interactions.

Chapter 4 briefly reviews Visual Interactive Modelling (VIM) and Visual Interactive Simulation (VIS) concepts and discusses their benefits in learning and teaching. We then argue the use of Adobe Flash and its scripting language to create a tool to support both concepts. Since VIS combines simulations and animations, some approaches for integrating these features are also discussed. How VIS's essential components can be created with the help of our framework is then presented. We subsequently present how a series of our simulation components can be used to build queuing models. This chapter ends with a discussion of some tricky issues in integrating an animated simulator to DES models specifically in permitting animation speed to dynamically be adjusted by learners during model runtime.

Chapter 5 discusses how to systematically design a tool for building attractive and interactive DES models. We first review component-based tool principles and examine how these principles can ease model building. We then suggest the Delegation Event Model for forging links between DES active and passive components. Next, we present the MVC (Model-View-Controller) pattern and discuss how it can be utilized for loose coupling between components, their interfaces (GUIs) and their visualizations. We further our discussion on how to cater with

model complexity through model partitioning (i.e., hierarchical model development) and how to support such development using the two patterns.

Chapter 6 reports two experiments that collected users' feedback of the tool and its resulting models. One experiment evaluated learners' perceptions about the attractiveness and interactivity of the models. We developed our own questionnaire for this based on model features proposed by relevant studies. Another experiment evaluated model designers' perceptions about the perceived usefulness, perceived ease of use and perceived enjoyment and their willingness to use the tool in the future. For this, we used the Technology Acceptance Model (TAM) and other extension models found in the literature. We also assessed the participants' workload while experiencing our tool using NASA Task Load Index (TLX).

Last chapter, i.e., *Chapter 7* concludes the findings of the research, lists some of its limitations and proposes some recommended future work.

CHAPTER 2

SIMULATION AND EDUCATION

2.1 Introduction

Many studies (e.g., Charles, 2008; Kauchak & Eggen, 2007; Nigel, 2008; Wurdinger & Carlson, 2010) argue in favour of blended learning, which mixes different learning environments (face-to-face and computer-based materials) and approaches to teaching a subject. Typically all of these require that teachers prepare a set of activities that support students' cognitive styles and make learning an engaging activity. Teaching approaches that are merely based on traditional lectures (which are typically constrained to one-way communication), static learning materials and individual or group assignments, will often result in only a shallow understanding of course contents and decrease students' motivation and enthusiasm for the taught subjects. Better approaches seek to engage learners' attention and actively involve them in the learning processes.

To make learning enjoyable, several instructional methods have been suggested; e.g., collaborative learning (i.e., a group of learners cooperate in their learning activities), problem-based learning (i.e., a group of learners collaboratively solve assignments with the help of a teacher) and computer-supported instruction, such as simulations and educational computer games. In this context *simulations* can act as important tools for discovery-based learning (Jong & Joolingen, 1998; Reid, Zhang, & Chen, 2003; W. R. Robinson, 2000; Zhang, Chen, Sun, & Reid, 2004) by offering a learning environment where learners learn by doing. Swaak and Jones (2001a, 2001b) suggest that simulations have three characteristics that enhance discovery-based learning; i.e.,

- *richness*, where knowledge is obtained through various dynamic representations such as animations and numerical data displays,

- *low transparency,* where knowledge must be inferred by learners themselves, and
- *active interaction,* where knowledge is obtained through experimentation.

Many benefits of including simulation models in teaching and learning environments as part of learning materials or as complementary activities have been listed in many relevant publications; e.g., by Aldrich (2004, 2005), Fitzpatrick (2003) and Gibson, Aldrich, & Prensky (2007). For example, active experimentation while exploring simulation models not only helps learners develop a mental model of real world processes or events, but can also support *collaborative* styles of learning (Beux & Fieschi, 2007; Jeffries, 2005) as well as *problem-based* learning through model building (Milrad, 2002). To fully reap those benefits, learners need models that demand hands-on interactions (to stimulate learning by doing) and offer support whenever it may be needed. How one can best integrate such models into appropriate approaches for knowledge construction and to enhance learning and problem solving skills has been investigated empirically by, e.g., Chang, Chen, Lin, & Sung (2008), Gokhale (1996), Kennepohl (2001), Liao & Miller (1996), Reid, Zhang & Chen (2003), Renshaw & Taylor (2000), and Rieber, Tzeng, & Tribble (2004).

This chapter examines the use of simulations in education. It scrutinizes different types of simulation models, their roles in education and learning, empirical evidence that establishes their effectiveness in e-Learning environments, some available simulation development tools, and current interests on animated DES models.

2.2 Simulation Models and Their Purposes

There are many different definitions of simulation. From an educational perspective, Castillo, Hancock and Hess (2004) and Aldrich (2002, 2004, 2005) define simulation as *digital learning material that allows learners to perform hands-on activities* (e.g., mouse clicking, text entering, etc.) in order to receive additional tasks or information. From an engineering perspective, the term refers to a model which *replicates a system's characteristics and behaviour based on specified goals of a study* (Flynt &

Vinson, 2005; Law, 2007; Shannon, 1998). Since they permit learning through experimentation in a safe and effective environment, simulations have become an increasingly popular educational tool and have been used for four purposes; i.e., to:

- *train learners' technical skills* or to demonstrate and practice tasks that are too dangerous or impractical to be performed directly; e.g., surgery or operating dangerous equipment. Since real environments are replaced with safe and cheap simulated environments, learners can freely explore their ideas, run a series of actions and examine the consequences. Such virtual environments not only reduce costs, but also offer learners the freedom of deciding when and where they want to learn.
- permit learners to *practise decision making in situations where proposed actions cannot be directly and immediately observed,* for example because their effects are delayed in time or/and dispersed in space. Since simulations can represent such situations in attractive and interactive forms and give feedback from learners' actions (i.e., allow them to stretch or compress time and space), learners can become more engaged and their learning experiences may be enhanced. Simplification while maintaining a high degree of fidelity is an important challenge for this use of simulation (Aldrich, 2004, 2005; Lunce, 2006).
- *explain concepts and complex interrelationships between variables*; e.g., in economic or queuing systems. In the traditional learning approach, teachers can only discuss complex interrelationships in verbal or textual forms. Watching models in execution and interacting with them can, however, lead to better and deeper levels of understanding.
- *provide learners with a diversity of "soft skills"* (Aldrich, 2005; Gaffney, Dagger, & Wade, 2008; Maldonado, Lee, Brave, Nass, Nakajima, Yamada, Iwamura, & Morishima, 2005; Vries, 2004); i.e., personal attributes (e.g., responsibility, common sense, motivation, etc.) that enhance an individual's interactions, job performance and leadership. Learners can use relevant models to practice a range of skills before applying them to the real world.
- *enhance materials to increase learners' motivation to learn a subject* (Castillo et al., 2004; Prensky, 2001). It has often been claimed that learning by doing can cause knowledge to be retained longer compared to just reading static materials in

traditional classroom settings. Within this context, simulations can offer more engaged and immersive learning materials for learners to learn about events, processes and activities.

2.3 Types of Simulation Models

We can classify simulations into three categories:

1. *live simulations* (or *role playing*), where real people manipulate simulated operations of a real system using real equipment (e.g., a training exercise of a fire drill),
2. *virtual simulations,* where real people operate simulated equipment in a simulated environment (e.g., flight and vehicle simulators), and
3. *constructive simulations,* where real people operate computerised models from which they obtain feedback.

While virtual simulations are well suited for some types of training, constructive simulations can move beyond simple rehearsal of skills and provide bases for easily analysing and comparing effectiveness and consequences of a wide range of physical or cognitive tasks. Thus, constructive simulations have long been used in a variety of domains in education. These include computer sciences (Aubidy, 2007; Yin, Ogata, & Yano, 2007), engineering (Ledin, 2001), logistics (Ganapathy, Narayanan, & Srinivasan, 2003), biology (Keen & Spain, 1992), medicine (Hoppensteadt & Peskin, 2002), economics (Porter, Riley, & Ruffer, 2004), physics (Chang et al., 2008; Jong et al., 1999), management sciences (Pidd, 2004) and sociology (Halpin, 1999; Moretti, 2002). Constructive simulations can generally be classified on the basis of the degree of learning support they offer: *single concept*, *operational level* or *strategic level*. A description, some characteristics and examples of each type of constructive simulations are shown in Table 2.1.

Table 2.1 Classification of Constructive Computer Simulations

Learning Support	Description	Characteristic	Simulation Type
Basic concept	• Simulations dealing with a simple calculation or a specific aspect of theory • Model behaviour is not affected by time	• Encourages learners to apply previous knowledge • Often found in educational simulations	Soft skill simulations Procedural simulations
Operational level	• Simulations dealing with specific operations • Model behaviour is changing at discrete points in time	• Stimulates students to explore, experiment, predict and invent given phenomena • Often found in engineering and science studies	Discrete Event simulations; e.g., queuing networks, manufacturing, logistic, etc.
Strategic level	• Simulations dealing with complex natural processes • Model behaviour keeps changing over time	• Provoke systemic thinking about given phenomena • Often found in engineering and science studies	Continuous simulations; e.g., biology, ecology, economics, sociology, etc.

Based on this classification, Chwif and Barretto (2003) have argued that those that support *operational* or *strategic* levels are more effective but difficult to design than simulations that those intended to simply train people in basic (e.g., device simulations for training operators of industrial machinery (Kaye & Castillo, 2003)) or "soft skills" (e.g., teaching skills in communication, leadership or strategic thinking (Gaffney, Dagger, & Wade (2008)). Table 2.2 shows how different types of simulations can be used to support learning in different domains.

Table 2.2 Simulation Types and Learning Support

Type	Learning support	Example
Soft skills simulations (also known as branching stories or situational simulations) (Gaffney et al., 2008; Idrus, Dahan, & Abdullah, 2009; Radcliff, 2005)	• Exposing learners to simulated work experiences in order to improve their communication and decision making skills *before* dealing with real situations • Exploring alternative paths through a task with additional information and instructions, based on learners' responses	• Software usage simulations • Situation-based simulations; e.g., in business and management training, customer and sales training, customer service training, doctor-patient interaction, etc.
Procedural Simulations or Virtual products (Kaye & Castillo, 2003; Michelson & Manning, 2008)	• Understanding the physical characteristics of real equipment • Learning to use costly equipment or perform complex tasks	• Mechanical device simulations; e.g., medical, manufacturing, home electronic equipment, etc.
Discrete Event Simulations (Banks, 1998; Wainer & Mosterman, 2010)	Understanding the operation of a system that traces ordered sequences of events	• Queuing systems • Manufacturing systems • Logistic systems; e.g., warehouses, ports, airports etc.
System Dynamics (Hannon, Ruth, & Meadows, 2001; Sterman, 2001)	Understanding the behaviour of systems that contain feedback loops involving stocks (entities that accumulate or deplete quantities over time) and flows (rates of change)	• Policy analysis and design • Population systems • Ecological systems • Economic systems

Alternatively, Castillo, Hancock and Hess (2004) divide educational simulations into two basic categories: *structured simulations* and *open-ended simulations*.

1. *Structured simulations* are used to support the understanding of system behaviour. Information is presented in a step by step fashion, where each step requires learners' responses to progress to the next of a number of alternative steps. Since

information is only delivered when requested, this approach has been claimed to *enhance* traditional learning styles. It can be used in all learning domains.

2. *Open-ended simulations*, on the other hand, leave learners to freely explore a simulated environment; this is typical for DES and system dynamics simulations. Although some studies (e.g., McKenna & Laycock, 2004; Michael, 2000) claim that there is no clear benefits of using open-ended simulations, other studies (e.g., Jong & Joolingen, 1998; Land, 2000; Lunce, 2006) have argued that great benefits can indeed arise from the fact that learners are *not* supported by additional instructions to overcome problems. This may forces learners to adopt a scientific discovery style of learning; e.g., by performing experiments. Opponents to this approach, however, assert that most students are unlikely to plan such experiments carefully enough, do not have sufficient skills to interpret outputs appropriately (particularly if models contain stochastic effects), and teachers may not state objectives or the learning outcomes clearly enough.

By looking at the way in which simulation impacts students' learning, Sahin (2006) clarifies the pedagogical difference between the two above mentioned approaches further. He distinguishes between *instructive* and *constructive* strategies.

Instructive strategies only consider learners as passive entities. They simply *consume* information with multimedia support. Such consumption and a limited degree of interactions can lead to some learning, but the impact on students' problem solving skills may be minimal. This is the case in structured simulations.

Constructive strategies meanwhile permit learning by freely exploring the relationships between a system's inputs and outputs through feedback obtained from a model. This is the case in open-ended simulations. The two-way interaction between experimentation and observation challenges learners' thinking and may eventually lead to acquisition of higher order thinking skills. Since such simulations are typically based on models of complex real-world systems, the knowledge or experiences gained from these interactions can later be transferred or applied to real-life scenarios. To make them effective, such simulations require some pre-knowledge; i.e. a basic understanding of the modelled systems. This must be supplied by teachers or appropriate instructions (Land, 2000; Min, 2003).

To either complement traditional classroom teachings or support distance education within a learning environment, two types of constructive simulations have been proposed by Neumann, Page, Kreutzer, Kiesel, & Meyer (2005). These are:

- *Simulation-based learning*; i.e., computer simulations are *used* to explain complex systems. To support knowledge acquisition through simulated systems, the combination of simulation, animation, visualization and various other instructional techniques is crucial.
- *Simulation-focused learning*; i.e., computer simulations are the vehicle through which all learning occurs. In this approach all related modelling concepts and methodologies are explained in detail, which then enables learners to apply simulation to practical problems. Simulation-focused learning is usually found in engineering and science courses.

2.4 The Role of Simulations in Education and Learning

2.4.1 The Role of Simulations in Learning Theories

In order to prepare suitable learning materials for learners, an understanding of the learning process is required. A *learning process* involves three main aspects: cognitive, emotional and experiential (Illeris, 2000; Livesey, 1986). The explanation of how these three elements shape learning is called a *learning theory*. Learning theories can be categorized into three main groups: *behavioural*, *cognitive* and *constructive* (two categories that will not receive further mention are *andragogy* (Knowles, 1984) and *connectivism* (Siemens, 2005)). Learning theories are used as a guidance to design and prepare learning materials based on learning goals and outcomes, and the format and contents of learning materials must assure the desired effects on learners' performance (R. C. Clark, Nguyen, & Swelle, 2006).

Behaviourism only considers observable aspects of learning processes (i.e., by observing changes in learners' responses), without allowing any speculation about processes that may occur in the learner's mind. Its main principle is that learning takes place through *repetition* and *reinforcement*. Continuous reinforcement (i.e., by

penalizing undesired behaviour while rewarding desired behaviour) is used to promote learning, while intermittent reinforcement seeks knowledge retention. While such strategies have been quite popular in conventional classroom teaching, they have proved only effective for teaching simple tasks. Common applications include taking, reading and memorising notes, and recalling knowledge and skills in tests or examinations. An example of educational technologies based on this theory is drill and practise software that delivers contents in small steps, with simple control questions at the end. Such software relies heavily on right and wrong answers, where "right" answers lead learners to new information, while "wrong" answers spawn repetitions. Since the Behaviourist theory does not explain learning and has failed to help understanding and acquisition of complex scenarios and skills, educators have looked for alternatives and *cognitive* theories, which attempt to take account of what may take place in a learner's mind.

Cognitivism asserts that the ability to construct new knowledge is strongly influenced by how well individual learners' memory can map (structure) new information to already acquired information. The new information (retained in a new logical slot) is then retrieved and modified to help process further new information. Each learner may have a different capacity for processing, retaining and using information. In order to ease the process of integrating new knowledge into existing cognitive structures, learners must have acquired all pre-requisite lower-level information before being exposed to higher-level concepts. An example of educational technologies based on this theory is an Intelligent Tutoring System (ITS), which guides learners throughout their learning processes.

Constructivist theories strongly emphasize the importance of prior knowledge, and view learning as a process of *actively* constructing new knowledge based on three elements: *prior knowledge*, *activities* and *experiences*. Active knowledge construction means that learners themselves are responsible to use and explore interactive learning materials and make use of all feedback to develop their mental models. These iterative processes are supposed to promote active learning (i.e., learning by doing) and extend knowledge retention. Since each learner differs from others in terms of pre-knowledge, experiences and relevant skills, the same learning materials will result in different knowledge structures and problem solving skills for different learners. An example of educational technologies based on this theory is simulation. Table 2.3 shows some features of learning theories.

Table 2.3 Some Learning Theories and Their Features

Feature	Behaviourist	Cognitivist	Constructivist
Learning principle	Observable objectives	Problem solving	Knowledge construction
Learning focus	Reflection	Cognition	Interaction
Teacher' s task	Transmitter	Helper or tutor	Facilitator
Instructional Design	Direct instructions, course based measurement, sequenced tasks	Problem solving through exploratory learning, project-based works	Self-directed learning, case-based learning
Learning material presentation	Linear contents that move from simple to complex	Dynamic, complex environments	Dynamic, unstructured (not pre-specified)
Human brain	Passive knowledge container	Linear information processor	Closed information system
Learning direction	Controlled by teachers	Controlled by learners with proper guidance from teachers	Controlled by learners
Learning outcome	Predetermined and predictable	Predictable	Unpredictable, since instructions only foster, not control learning processes
Evaluation	Performance based on correct answers where each unit of content is treated and evaluated separately	Knowledge based on discovering correct methods for finding answers	Competence (degree of mastery) based on dealing with complex problems
Learning measurement	Easily measured by counting correct answers	Indirect, based on active problem solving	Not easily measured and much more subjective, usually based on on-going activities, experiences and attitudes; e.g., notes, drafts, journals or products
Learners' interaction	Simple interactions with controlled presentation via verbal or graphical instruction	Demands intelligence	Demands more communicative and immersive contents to show how a model responds to individual assumptions through feedback

Criteria	Closed system where learners are confined to the teachers' world	Solution-centred	More dependent on learning than teaching approach
Knowledge construction	Encourages surface learning	Creates deeper problem understanding	Promotes deeper, transferable understanding and long term retention of knowledge
Software development time	More quickly constructed	Time consuming	Time consuming and much effort is needed, since it requires a significant amount of interactive and unstructured learning materials
Knowledge retention	Works well for short-term transferable knowledge	Better at long-term knowledge retention	More long-term and applicable, since knowledge is obtained through interactions and activities (leaner-learner and learner-model), not through competition among learners
Software characteristic	• Rigidly structured • Dearth of content interactions and forms of presentations • Sequential exposition of information, followed by testing	• Intelligent sequence • Modestly interactive. pre-packaged problems	• Unstructured, no pre-packaged problems, highly interactive • The use of animation and multimedia environments is common
Ideal software	Drill and practise programs, programmed instructions and tutorials	Intelligent Tutoring System, Computer Based Training	Simulations, microworlds (L. P. Rieber, 1995), modelling environment, hypermedia

The development of e-Learning materials based on cognitive and constructivist theories is an important step towards better learning environments since:

1. curricula are now packed with many subjects and learner-teacher interactions are limited,
2. the demand for education keeps rising, but time and space remain restricting factors, and
3. learners are now familiar with modern technologies and expect their use. Interactive contents therefore become crucial.

These factors favour a shift of responsibility from teacher-oriented (the behaviourist feature) to learner-oriented learning styles (the constructivist feature). Some approaches to transfer such responsibility are through *guided discovery* (R. E. Clark, Yates, Early, & Moulton, 2010; Leutner, 1993; Piaget, 1977), *case-based learning* (Aamodt & Plaza, 1994; Jonassen & Land, 2000) and *microworlds* (Brouwer, Muller, & Rietdijk, 2007; L. P. Rieber, 1992, 1995, 1996).

Guided discovery enables learners to create their own understanding of a subject, using tools (e.g., simulations) with guidance from a teacher. Since the role of a teacher changes from a transmitter of information to a promoter of higher-order thinking skills, this method has been claimed to be an ideal approach in education (Aldrich, 2004; Chwif & Barretto, 2003; Gibson et al., 2007; Gokhale, 1996) and is believed to produce "deeper" learning than teacher-centred approaches (e.g., demonstration, direct instructions, lectures or lecturer-discussion).

The main strength of simulations in this context is that it enables a "situated learning" approach (Der-Thanq & David, 2002; Herrington & Oliver, 1995, 1997), which claims that realistic contexts will motivate learners to engage more strongly with the material. Since this instructional methodology requires learners to be equipped with a substantial amount of pre-knowledge and skills, several studies (e.g., Kirschner, Sweller, & Clark, 2006; Tripp, 1993; Wineburg, 1989) criticize its implementation in traditional classrooms. However, some other studies (e.g., Harley, 1993; Ketelhut, Dede, Clarke, Nelson, & Bowman, 2007; Lunce, 2006; Young, 1995) report strong support for embedding situated learning through use of modern educational technologies.

Simulations can be used in a variety of learning and training domains, since most aspects of real-life processes and job environments can be simulated in controlled settings. Simulations are appropriate for teaching situations when learners

can gain high levels of knowledge and skills (i.e., application, analysis, synthesis and evaluation levels in Bloom's taxonomy - Krathwohl, Bloom, & Masia (1996), Anderson & Krathwohl (2000)). However, since simulations are associated with constructivist learning theories, they have some disadvantages, which include:

- Simulations heavily depend on learners taking responsibility for their own learning. Without motivation, significant learning will not take place.
- Simulations require tacit knowledge and particular skills to enable learners to drive experiments, analyse and understand feedback, draw their own conclusions and predict a chain of actions throughout a learning activity (Whiteside, 2002).
- Simulations demand coaching and scaffolding to offer learners hints at certain times (Min, 2003; Zhang et al., 2004). Without these elements, learners might interact with simulation models without framing sensible hypotheses and may draw wrong conclusions. However, too much guidance will stifle learners' creativity, since they are now confined to a series of tasks (Herrington & Oliver, 1995, 1997).
- Simulations need collaboration (i.e., learner-learner and learner-teacher discussions) to promote critical thinking and problem solving skills.
- Simulations may require more time for learners to abstract meaningful knowledge, since learners need time to immerse themselves into a problem and experiment with alternatives (Heinich, Molenda, Russell, & Smaldino, 1999).
- If they are overly simplistic, simulations may create an imprecise understanding of real-life situations.
- Simulations need tools that offer authentic contexts and activities (Herrington & Oliver, 1995, 1997; Lloyd P. Rieber et al., 2004) to engage learners' attention. Authentic contexts reflect how knowledge can be used in real-life and motivate learners to use the model. Authentic activities ask learner to find and solve problems themselves. Thus, explorative models that allow manipulation of widest ranges of variables are crucial to stimulate learning by doing (Kolb, 1984; Whiteside, 2002). However, designing, building and testing such simulations is time consuming and costly.

2.4.2 Empirical Evidence

Simulations have long been claimed to have positive effects on learning (e.g., Lunce, 2004; Min, 2003; Njoo & Jong, 1993; L. P. Rieber, 2002). Some researchers have conducted experiments to evaluate the effectiveness of using simulations either as complement to or as a replacement for other learning materials and tools. Such studies include Liao and Miller (1996), Gokhale (1996), Michael (2000), Renshaw and Taylor (2000) and Kennepohl (2001). Eck and Dempsey (2002) meanwhile have examined the impact of embedding advisement and competition in computer simulations.

Liao and Miller (1996) have studied the effects of using computer simulations as complementary learning materials on learning in a construction and architectural engineering technology course. Analysis of the course examination results showed that the mean and median for the group supplied with both text-based course materials and a simulation game was higher than the group supplied only with the text-based materials, supporting the thesis that a computer simulator as a companion to reading materials could help learners learn better.

Gokhale (1996) has examined the effectiveness of using computer simulations to teach problem-solving skills in an electrical course. Data analysis showed that students exposed to a computer simulation in addition to lecture-lab activities were significantly better than students that only used traditional lecture lab activities. The results therefore corroborated the assumption that simulations could be an effective learning approach to equip students with problem-solving skills that are transferable and applicable to real world problems.

Michael (2000) has explored the possibility of using a computer simulation as a replacement for real-hands-on activities in creating a product. They found that no significant difference in product creativity scores among the hands-on group and the computer simulation group. This suggests that it was possible to use a computer simulation in place of hands-on activities while maintaining student creativity.

Renshaw and Taylor (2000) assessed the impact of using system dynamics simulations on students' higher-order cognitive skills of environmental processes. Data analysis showed that the students who had been exposed to the simulation had a better understanding of what they had learnt (i.e., simulation had a positively impact on students' higher-order cognitive skills), were less prone to cognitive errors in

decision making and demonstrated higher information retention compared to the students who obtained equivalent information through hands-out.

Kennepohl (2001) examined the effectiveness of simulations in enhancing students' chemistry laboratory experiences. They found no difference in overall course performance between the students who fully attended supervised laboratory sessions and the students who were partially attended the sessions but were additionally exposed to individual laboratory simulations. However, the later students completed laboratory work in a shorter time and achieved a slightly better performance in the practical laboratory component (lab reports and quizzes). This suggests that computer simulations can enhance student lab experiences in spite of lab time reductions.

Eck and Dempsey (2002) have studied the effect of embedding advisement and competition elements in a computer-based simulation to teach the concepts of geometric shapes. Advice through interactive videos could be accessed whenever students were stuck at certain problems. Competition refers to whether or not the students were playing against computer characters to encourage their learning. The results showed that (1) the presence of advisement during simulation did not guarantee to help learning unless it was properly designed and used, (2) advisement was probably effective in promoting learning in a leisure environment, (3) the presence of advisement during competition could create additional cognitive load and hinder learning, and (4) knowledge transfer could be promoted as long as there was a connection between the learning context and students' prior knowledge no matter which approach was used.

2.4.3 Simulations and e-Learning

2.4.3.1 Promises and Problems of e-Learning

e-Learning utilizes electronic documents for facilitating learning. It has been boosted by *globalisation* that forces people to regularly update their knowledge in order to compete in the current job market, *technological improvement* particularly in *software* that simplifies the development of attractive and interactive learning materials for

better learning experiences and *internet speed* that eases remote storing, updating and accessing of the materials.

Recent research clearly shows the growth of e-Learning in educational institutions and training organizations worldwide to support traditional classrooms and/or offer virtual learning environments (Ala-Mutka, Gaspar, Kismihok, Suurna, & Vehovar, 2010; Garrot, Psillaki, & Rochhia, 2008; K. Kim, 2006). This type of learning has been accepted as a typical teaching and learning platform since the development of learning management systems (LMSs) that offers various learning supports through the use of current technologies (e.g., online assessment, communication, etc.) and the familiarity of current learners with a self-directed learning environment through the use of computer. The use of e-Learning as a virtual learning environment through the support of information and communication technologies (ICT) can promise:

- Learning anytime, anyplace. Learners can study learning materials without time constraints. This gives learners opportunities to learn and access a much wider range of knowledge. Study can take place either at home, work, libraries, etc. as long as learning materials can be accessed.
- Collaboration through synchronous and asynchronous interactions. This enables learners and teachers to discuss and exchange information at anytime and anywhere. Such facilities are available in most LMSs.
- Learning through new technology approaches. Current learners are computer-literate and familiar with learning through computers. These opportunities can be utilized by e-Learning content designers to provide highly motivating attractive and interactive styles of presentation; e.g., interactive simulations and computer games. Such methods when used properly are claimed to engage learners, enhance e-Learning experiences and decrease the amount of reading, which improves the retention of the materials (Aldrich, 2004, 2005; Neumann et al., 2005).
- Cost effective. The use of technology can reduce costs related to teachers, physical spaces, hardcopy of learning contents, etc. Learning can be delivered on time.

Improving the quality of e-Learning experiences remains a continual challenge for LMSs. Most e-Learning materials have been constructed without much consideration of how learners learn (Dublin, 2004; McKenna & Laycock, 2004; Romiszowski, 2004) where the use of static graphics (e.g., e-book, Word documents, etc.) and simple online assessments (e.g., simple multiple-choice and true-false recall type of questions) is common (Neumann et al., 2005; Wahlstedt, Pekkola, & Niemelä, 2008). These materials cannot be considered quality e-Learning solutions since they only deliver facts and fail to engage and attract learners. Consequently such materials typically fail to promote a constructive and cooperative learning style and fail to facilitate the transfer of knowledge to job environments; i.e., the utilization of the knowledge (Kühl, Scheiter, Gerjets, & Gemballa, 2011; Wilson, Jonassen, & Cole, 1993). The importance of interactivity, visual presentation and aesthetics in learning materials has long been suggested in the relevant literature (e.g., Bransford, 2000; Eppler & Burkhard, 2007; Mildrad, 2002).

2.4.3.2 The Roles of Electronic Course Management Systems

Most educational institutions and training organizations now support teaching and learning activities with LMSs. LMSs (also sometimes called *Course Management Systems* (CMSs)) offer tools for both management and delivery of course materials and assessments. Open source LMSs include *Moodle* (www.moodle.org) and *.LRN* (www.dotlrn.org). Other LMSs, such as *WebCT* (www.webct.com), *Blackboard* (www.blackboard.com) and *eCollege* (www.ecollege.com) are sold as commercial products. The roles of LMSs are to:

- provide content management through attractive GUIs and layouts in order to ease store, structure and distribute learning materials. Such characteristics are important to foster a pleasant experience when using and learning through the platform (Stenalt & Godsk, 2006).
- provide advanced communication facilities through synchronous and asynchronous modes. The *synchronous* mode tries to imitate traditional learning environments and assumes that a group of learners and their teachers will be

online at the same time. It uses chat rooms or video conferencing technology as a communication. In contrast to this approach, an *asynchronous* mode that uses email and bulletin board allows each learner to be online at times that depend on his or her preference.

- track learners' behaviour and performance, and record the number of times learners access certain content, as well as the time spent on studying different content materials. In order to support this communication, learning materials must comply with a set of technical standards for e-learning; e.g., SCORM (Gonzalez-Barbone & Anido-Rifon, 2010; Vossen & Westerkamp, 2006).

The development of LMSs to support virtual learning and teaching activities has increased the use of e-Learning in higher education institutions worldwide (Browne, Jenkins, & Walker, 2006; Falvo & Johnson, 2007). However, providing right learning materials (based on learning pedagogy) and supporting them through various learning facilities available in LMSs are important in promoting student involvement and ensuring the success of e-Learning (Klobas & McGill, 2010).

2.4.3.3 Pedagogical Aspects of e-Learning

e-Learning shifts the medium of knowledge and skill transfer from a teacher to computer. This transfer should imitate whatever important features in the traditional classrooms (e.g., activities that involve learners in the learning processes, two-way communication that allows learners to respond and get feedback, etc.) and incorporate them all into the virtual learning environment (Alonso, Lopez, Manrique, & Vies, 2005). The absent of teachers during learning time must be replaced with new methods of instruction design that stimulates student engagement and involvement. Instructional methods that are based on attractive and interactive materials (e.g., simulation, computer games, etc.) and that provide activities that will impart learners' knowledge and skills are important in guaranteeing successful learning outcomes.

Attractive and interactive materials that are based on dialoguing, controlling, manipulating, searching and navigating (Moreno & Mayer, 2007) play three important roles in virtual learning. Firstly, they can replace the dialogues between learners and

their teacher and promote motivation for them to learn through multidirectional communication (i.e., actions and feedback). Thus, learners will not be bored as reading static texts, viewing static graphics or navigating non-interactive materials (e.g., a narrated representation with animation, hypermedia, etc.). Secondly, they can stimulate information acquisition and knowledge construction (Fletcher & Tobias, 2005; Moreno, 2006) especially if they are designed to support *different modes* of presentation; e.g., verbal explanations (e.g., printed words, spoken words) and non-verbal (e.g., animation) and *mixed-modality* representations (i.e., auditory and visual). The approach of using multiple representation to illustrate content of knowledge eases learners to utilize knowledge and enables meaningful learning to occur in their cognitive (Moreno & Mayer, 2007). Thirdly, they stimulate meaningful communications among learners and increase the use of communication facilities provided by the LMSs to a maximum level since their activities will challenge learners' understanding during their learning activities. If the given outputs contradict with their hypotheses, learners will seek clarifications from their peers or teacher.

Attractive and interactive learning materials however do not automatically create understanding. Besides their effectiveness depends on learners' prior knowledge and their cognitive factors (Kalyuga, Ayres, Chandler, & Sweller, 2003), the interactivity could also create the potential of cognitive overload that disrupts learning (Mayer & Moreno, 2003). Thus, it is important to design learning materials that (1) manage the amount of information presented at a time, and (2) reduce *extraneous processing*, i.e., the cognitive processes that add burden to digest new information (e.g., asking learners to refer to information in other pages or computer screens) and *representational holding*, i.e., the cognitive processes that force learners to hold their mental models during the making process (e.g., presenting animation after narration) that waste learners' cognitive capacities. For this, Moreno & Mayer (2007) propose instructional design principles for interactive learning materials. The design principles are *guided activities* to guide learning, *reflection* to encourage information acquisition, *feedback* to repair learners' misconceptions, *pacing* that enables learners control their learning and *pre-training* to provide learners with relevant prior knowledge.

2.5 DES Development Tools

Generally, DES models can either be built in general purpose programming languages, simulation packages, simulation languages or high level simulators as shown in Table 2.4.

Table 2.4 Available DES Simulation Tools

Tool	Example	Advantage/Drawback
Simulation Packages	**Non Object Oriented** CSIM (Schwetman, 1988), GASP (Rose, 1981), SimPack (Fishwick, 1992), SimTools (Seila, 1986), SIMPAS (Bryant, 1981).	Advantage: ▪ Reduce programming effort by providing simulation-specific features Disadvantage: ▪ Prone to logical and syntax errors ▪ Depend heavily on model developers' programming skills ▪ Do not usually offer animation capability
	Object Oriented CSIM19 (Schwetman, 2001), C++Sim (Little & McCue, 1993), DESMO-J (Meyer et al., 2005a), JavaSim (the Java version of C++SIM) (Tyan, 2002), JSIM (allow simple VIM) (J. A. Miller et al., 1998), J-Sim (Kacer, 2002), PSim (Garrido, 1999), Silk (Kilgore, 2000), simJAVA (W. Kreutzer, J. Hopkins, & M. C. Mierlo, 1997), Simjava (E. H. Page, Moose, & P.Griffin, 1997), SimKit (Buss, 2002), Sim++ (based on SimPack) (Lomow & Baezner, 1989), SSJ (L'Ecuyer et al., 2002).	
	Object Oriented and support animations D-SOL (Jacobs, Lang, & Verbraeck, 2002), Tomas (Duinkerken, Ottjes, & Lodewijks, 2002; Veeke & Ottjes, 1999), Psim-J (Garrido, 2001, Garrido and Im, 2004).	
Simulation Languages	**Non Object Oriented** GPSS/H (Crain & Henriksen, 1999), SIMAN (C. Dennis Pegden, 1989), SLAM (Claude Dennis Pegden, Alan, & Pritsker, 1978), SLAM II (Pritsker, Sigal, & Hammesfahr, 1994), SLX (Henriksen, 1997)	Advantage: ▪ Offer much flexibility for simulation model development Disadvantage: ▪ Still need substantial programming expertise
	Object Oriented SimPy (Matloff, 2008), SIMSCRIPT (Markowitz, Hausner, & Karr, 1963; Rice, Marjanski, M., & Bailey, 2004), SIMSCRIPT II.5 (Kreiman & Mullarney, 1987), SIMSCRIPT III (Rice, Marjanski, Markowitz, & Bailey, 2005), Simula (Birtwistle, 1979), MODSIM III (Goble, 1997).	
High Level Simulators	baseSIM, Extend (Krahl, 2003), ExtendSim7 (Krahl, 2007), SIMUL8 (Concannon et al., 2006), AweSim (based on SLAM II) (O'Reilly, 2002; Pritsker & O'Reilly, 1999), Micro Saint (Barnes & Laughery, 1997), Arena (based on SIMAN) (Bapat & Sturrock, 2003; Kelton et al., 2004), WITNESS (Thompson, 1996),	Advantage: ▪ Easier to learn ▪ Speed up the model building process and the analysis of model output ▪ Much simpler to

	Promodel (Harrel & Price, 2003), AutoMjod (LeBaron & Jacobson, 2007), Flexsim (Nordgren, 2003), SIMPROCESS ("Getting Started with SIMPROCESS," 2006), Renque ("Renque Discrete Event Simulation: User's Guide," 2008), em-Plant ("m-Plant: Empower for Manufacturing Process Management," 2003), Simple++ (Geuder, 1995), SIMFACTORY II.5 (Goble, 1991)	maintain and change compared to simulation languages or simulation packages ▪ Can incorporate sophisticated animations to depict system behaviour Disadvantage: ▪ Commercial tools are expensive to buy and not so flexible

General purpose programming languages (e.g., C, C++, Java, etc.) allow greater programming flexibility, but require model developers to be expert in a particular programming language. Since models are developed from scratch, they take a longer time to be built and are prone to syntax and logical errors. Developing DES models using this approach is far from ideal in learning and teaching environments, since both teachers and students typically need easy tools to quickly build and animate a model's inner working.

Simulation languages allow simulation models to be developed using customized modelling statements. In spite of their strength in modelling almost any kind of complex system, a modeller still needs programming expertise, as well as knowledge of their specific features (e.g., linguistic abstractions) and representation of model logic. Although most simulation languages support animation, the resulting models often do not allow interactions and cannot be embedded on web pages or be integrated with e-Learning systems.

High level *simulators* allow models to be constructed by dragging and dropping readymade blocks onto a canvas. These blocks are then linked with each other through pads (input and output points) using connectors. The use of blocks to represent model logic facilitates model building and decreases model development time. However, the manipulation of models is only allowed through whatever features the package provides. Although most high level simulators support animation in 2D or 3D, the models can only be run in the system itself or by using the system's player. Few of them can be embedded in web pages.

2.6 Animated DES Systems

DES models are implemented as sets of computer codes that represent their relevant complex system processes' evolution through time. In this context, animations are used to gain insight into the systems through animated scenarios or graphical displays of statistical measures. Visually accurate animations can be crucial for better understanding of the models.

The benefits of animated DES models have been extensively discussed in the literature (e.g., Belfore, Mielke, & Kunam, 2003; Gilman, 1985; Hill, 1996; Kamat & Martinez, 2007; Kelton, Sadowski, & Swets, 2010; Macal, 2001; Rekapalli & Martinez, 2007; Stahl, 2003; Wenzel & Jessen, 2001). An animated model can:

- present its simulation processes in a more user-friendly and more easily understood form than textual traces of event sequences to improve users' understanding of a system
- clearly illustrate its structure and logic and allow users to visually study and analyze its process flows
- assist model developers in debugging (correcting syntax and logical errors), verifying (checking whether the model is functioning as intended) and validating (checking whether the model reasonable represents a real system being modelled) the model
- make simulation results more comprehensible, which aids the analysis of simulation results to gain better understanding of system performance under various conditions
- give insight into model behaviour during a simulation run in addition to numerical and statistical analyses at the end of a simulation run

Animations to improve the display and analysis of model execution are considered a significant augmentation of DES methodology, caused by a shift towards graphical model building and process orientation in modelling worldviews (Pedgen, 2007). New simulation tools that incorporate high quality 2D animation (e.g., Arena or ProModel) or 3D visualization (e.g., AutoMod, QUEST or eM-Plant) capabilities

are preferred to older tools that do not offer such capabilities (e.g., SIMAN and MODSIM). However, the high quality animations offered by these commercial tools fail to offer any means of interaction with their model; i.e., they do not allow users to change system conditions while the model is running. One of the reasons for this is a loss of execution efficiency, a consideration that is much less relevant in educational contexts than in DES technology's predominant commercial use for performance prediction.

Many researches that aim to add 2D or 3D visualization and animation capabilities to conventional simulation tools have also been many conducted (e.g., see Belfore et al., 2003; Kamat & Martinez, 2007; Zhong & Shirinzadeh, 2004). Most of them are based a *post-processing* approach that only enables an animator to enhance the visualization of objects, their states and behaviour *after* a simulation run. Moreover, model developers need to (1) learn how to use a particular simulation tool before generating customized simulation output files, (2) have enough programming knowledge to generate such files from within the model, and (3) modify the files; e.g., by inserting necessary commands for driving animations. Although this approach offers the capability to jump back and forth in simulated time during animation playback and to accelerate or slow viewing speeds, it is incapable of supporting runtime interactions with its animations.

Largely for marketing reasons, many simulation tools now focus on 3D visualizations since they promise to enhance presentation of simulation results. From a more practical perspective, 3D animations have not proved all that useful (Alam, Oloruntegbe, Oluwatelure, Alake, & Ayeni, 2010; Oloruntegbe & Alam, 2010) unless they are for simulators meant to train system operators (e.g., flight simulators). In other cases, 2D animation is usually adequate to capture essential system behaviour. Animations that offer interfaces that allow users to be animation directors (i.e., they can completely control each animated object rather than just viewing it, moving it, or changing its shape or appearance) are able to add more realism to simulated scenarios here. However, there must still be a clear separation of simulation and animation concepts.

Although not directly related to the mapping between a simulation model and its visual representation, Benjamin, Mazziotti and Armstrong (1994) suggest some significant requirements for offering attractive animation models. These include:

- appropriate icons or symbols with names that correctly represent the purpose of animated objects in a predefined library
- icons placed on an animation stage should have user-customizable label names to ease cross referencing and undefined icons
- statistical reports that can be customized with headings, labels, etc.
- graphical interaction windows for receiving input from users
- multiple windows to view information in different formats
- zooming ability to view details of a specific area of interest

While items (i) to (v) can be programmed, item (vi) places stricter demands on a programming language environment. It is therefore important to choose a language environment that supports the capability.

As stated, many researchers have investigated software that animates simulation results generated by separate simulation tools. This is a simplest way to graft animation capabilities onto existing systems. If no interaction is needed this may be a viable approach. However, such an animated model only suits users with concrete concepts of the represented system and typically fails to be used in a learning environment (Arbaugh & Benbunan-Fich, 2007; Su, Bonk, Magjuka, Liu, & Lee, 2005; Woo & Reeves, 2007). Thus, models for teaching and learning purposes should at least implement some kinds of interaction features to engage users and foster their learning.

Below are some attempts for connecting simulation and animation. Since the tools are separated, animated models based on this approach have two distinct limitations: (1) interaction features that allow two-way communication (i.e., animation that reacts to users' actions and any means that allow users to respond to model information) cannot be supported, thus users are constantly served with the same data driven animation, and (2) users are confined with static model graphical user interfaces as no visualization tools can be attached during model execution since simulation performance data is stored externally in the simulation tool.

Shi and Zhang (1999) create a platform for simulating and animating an activity-based model using simple 2D icons. In this context, models are built using activities blocks. Each block has its own dialog box for specifying its attribute values,

resource requirement, activity duration and an icon for presenting resources. Blocks are connected using an arrow to represent logical sequences of activities. To animate a resource's states, one or more pre-created bitmap icons can be chosen from a library, which stores common construction resources (e.g., trucks, cranes, etc.). During animation, icons move along specified paths and change shapes. However, animation of construction activities can only be performed *after* a simulation is finished. Although the tool does not allow user interactions with animated objects, the system offers some run-time control, such as starting and stopping a simulation and adjusting its animation speed.

Kamat and Martinez (2001) create a system called Dynamic Construction Visualizer (DCV) for animating construction operations in a 3D virtual space. The system reads a trace as an ASCII text file, which contains commands such as PATH (for defining paths between two locations in 3D coordinates), CLASS (for importing a 3D file in VRML format that represents resources and system entities), TIME (for driving animations at appropriate times), CREATE (for creating simulation objects), PLACE (for placing objects at appropriate positions), MOVE (for objects that may encounter time delays) and ROTATION (for rotating objects along specified planes). This file can be generated manually or written by simulation software. At an appropriate simulation time, DCV reads and performs the commands to drive animation. Animation is stopped when no more statements are found in the file, or when a viewer interrupts the animation. DCV allows animation to be run at any speed.

Belfore et al. (2003) describe an approach for producing 3D visualizations that can be played in the form of VRML (a standard file format for presenting 3D objects in a web browser) animations. The VRML contains a VRML scene (background transformation), VRML nodes (3D animated object transformation) and simulation model information and results obtained from a simulation tool with added information to create and animate 3D worlds (e.g., position, path, etc.).

Zhong and Shirinzadeh (2004) create an analyzer to convert important processes in simulation models (developed using whatever simulation tools) to animation events. The analyzer will group a sequence of events into events that belongs to an object based on their source objects and the event sequence it participates in. Events that are not important (e.g., no change in an object's position) will be filtered out. Each object is firstly positioned at its proper location in a 3D

layout editor and is then animated based on its animation events using animation viewer.

2.7 Summary

Previous work on DES construction tools has simplified model building that initially demands a substantial of programming effort to model building that only requires dragging and dropping blocks of code. Approaches to connect DES models with animations and visualizations that help learners to get insight into the models' processes and behaviour by showing their sequences of events have also been proposed. At the same time, commercial software has provided excellent tools for modelling, animating and analyzing DES models. However, none of the current tools have considered how learners' learn. The main lesson from this chapter is that models for learning purposes should support runtime interactions since interactions through various engaging activities can help learners to construct and develop their mental models of a domain. Additionally, the models should have relevant features to help learners engage in their learning. Table 2.5 show the features identified from the literature review as being desirable for the design of DES tools.

Table 2.5 Desirable Features for the Design of DES Tools

Feature	Purpose
Illustration of model structures and logic	Help learners visualize process flows
Feedback and performance visualizations	Aid learners to gain better understanding of system performance
Activities through easy-to-access GUIs	Allow learners to input simulation parameters
Attractive animation of simulation processes	Facilitate learners to get insight into model behaviour and improve their understanding
Multiple visualization windows	Enable learners to view information in different perspectives
Appropriate symbols and names	Represent the function of animated objects
Top level control of simulations and animations	Provide learners a choice to control simulation speed
Zooming Ability	Offer learners to view details of a specific area of interest

These models should also be web based models so that they can be embedded in a web page and LMS-compatible models so that they can easily be integrated with an LMS to take full advantages offered by the system. The next chapter will discuss how to properly design DES tools for building informative interactive DES models (that contain interactive and attractive GUIs, statistical tables, information windows, animation control, etc.) that are ideal for learning and how Flash supports the development of the tools.

CHAPTER 3

A FRAMEWORK FOR DES AND ANIMATION

3.1 Introduction

Dynamic systems contain various time-dependent processes and interconnected elements. There are two techniques used to study and evaluate such stochastic time-oriented systems: *analytic* and *numeric*. While analytical models can offer accurate solutions, it is unpractical (and typically fails) to model systems with very complex structures. A numerical technique (e.g., simulation) that uses numerical approximation is always a choice.

Time-oriented simulation imitates a system's behaviour over a period of time. There are two types of simulations under this classification: *discrete event simulation* (DES) where state variables change values at discrete time and *continuous simulation* where state variables change values throughout time. The main advantage of using DES to analyze discrete event systems over analytical models is that we only consider elements and their interactions that influence the system's behaviour, based on the objectives of our study. Essential elements that simplify model development in many types of DES systems have long been studied and presented.

DES has two different purposes. One focuses on *decision making* where simulation is used as a *prediction tool* for estimating performances of limited, risky and costly systems. Thus, the quality of a simulation model is paramount for feasible predictions. For this, its modelling approach must go through a number of cycles: system identification, model design, data collection, model implementation, model verification, model validation, model experimentation and model output analysis. Model implementation involves a transformation of a set of system significant features to a computer program. Model verification ensures that the program contains no errors and logically represents the system in terms of its functionality and structures. Model validation ensures that the program reasonably represents the

system behaviour (up to a certain level of confidence) in terms of accuracy of outputs it generates. If both conditions are satisfied, the model can be used for exploration. This includes changing model parameters (e.g., random numbers of arrival, routing policy, priority rules, server scheduling strategies, etc.) and/or model structures to improve its performance. Detail explanations of the modelling cycles can be found in most DES textbooks (Banks, 1998; Garrido, 2001; Kelton et al., 2004; Law, 2007) with Law (Law, 2007) give detail explanations on simulation analysis.

Other focuses on *teaching* about complex (natural, organizational or technical) processes. Compared to the first purpose that focuses on a quantitative aspect, the second purpose focuses more on a qualitative aspect. In this context, a simulation model is mainly used as an *exploration tool* for gaining insight into a system; i.e., to help users to understand aspects that influence its behaviour and sensitivity. Thus, providing a graphical representation of its structures, any means for its parameter manipulations and facilities for observing the effect of the manipulations (preferably without re-running the model) to current simulation results (e.g., through animations and visualizations of its state values) are particularly useful in offering many cognitive advantages for achieving this purpose.

Both purposes require basic tools for model implementation (i.e., constructing and running simulation models). The only different is that the extension of the tool, where one stresses more on providing tools for statistical analysis while the other one stresses more on providing tools for structural and behaviour visualizations.

Developing simulation tools is not an easy task. It must be well designed and structured in a reliable fashion based on an appropriate framework for preserving its flexibility and extensibility. This framework consists of segments; each of which handles its own functionality and cooperates with each other to accomplish a further task. The segments are later translated into computer code (i.e., simulation libraries) that can be called, initialized and assembled to construct a model.

Although the *library-based approach* offers ease of coding, they only support model construction using text descriptions. Thus, a *component-based approach* that offers a drag and drop fashion for model building and GUIs for easy accessing libraries' parameters while still supporting API (Application Programming Interface) has been introduced. The use of relevant symbols to depict components' functionality have been proved to offer some advantages especially in visualizing model structures and processes (Repenning, Ioannidou, Payton, Ye, & Roschelle, 2001; Roschelle et

al., 1999). However, runtime experimentations through the symbols' parameter modifications and responsive animation and model visualization customization for observing the effects of the modifications are still uncommon. This chapter focuses on a framework that leads to the construction of our *component-based tools* for animated interaction-driven DES models.

This chapter starts with a brief introduction to DES and queuing networks. A good understanding of DES mechanisms eases the development of our DES tools. We first discuss basic mechanisms of two available DES modelling styles, i.e., *event-oriented* and *process-oriented* and their suitability in implementing a DES engine. Because of some limitations of the event-oriented, we have architected our own process-oriented DES framework to support various DES facilities (e.g., generating random numbers for various distributions, collecting statistics, managing simulation clocks, a list of future events, etc).

This framework has been designed so that a collection of classes for providing simulation libraries can be constructed easily using any programming languages. While there are many programming languages that can be used to implement this framework, the use of appropriate programming languages that offers a user-friendly environment, supports OOP and eases integration of animation (e.g., facilities for creating new images, importing outside images, attaching those images to classes and animating objects through built-in animation methods) is important to support its further extension and to guarantee users' acceptance and satisfaction. For these reasons, we argue that Flash is a suitable implementation tool for any kinds of simulations (details on this will be discussed in Chapter 4).

3.2 DES and Queuing Scenarios

DES is a mathematical model that operates a system using a chronological sequence of events; each of which happens at discrete time. The execution of each event (e.g., the arrival and departure times of customers in a service system) will update model states, advance model time and consequently lead to a new event. Anything happens between the two consecutive events are ignored since they will not affect model behaviour. The change of state values is used to calculate various system performances.

Such a computational mechanism can be found in a wide variety of systems. Examples include manufacturing, transportation, service, network, inventory and computer systems with the main focus is to study and analyse *queuing networks* that explore the effects of capacity constrained resources and routing strategies on common performance measures; e.g., the average waiting time in a queue, resource utilization, throughput, etc. Results from this can be used to manage queues especially in deciding scheduling strategies and the number of resources needed to provide particular services. Analyses of queuing networks using simulations can be found in much literature (e.g., Fan, 1976; Guan, Woodward, & Awan, 2006; Raatikainen, 1997; Zhuang, Wong, Fuh, & Yee, 1998).

DES is generally built up by objects known as *entities* that move through simulated time. There are two types of entities: *transient* and *resident*. Transient entities enter and depart from a system with relative frequencies and may seek for services. In other applications, they are sometimes called as tokens, jobs, transactions, temporary entities, etc. Examples include customers in a service system, parts in a manufacturing system, vehicles in a transportation system, etc. Resident entities stay in a system for limitless times. They may offer services for transient entities and are sometimes called as resources, servers, facilities, permanent entities, etc. Examples include workers, machines, etc. The interaction among these entities will create other concepts such as scheduling (the availability of resources), routing, sequencing (queuing discipline) strategies and buffers (waiting spaces).

Each entity performs an operation at a finite time (either constant or random) called an *activity*. Activating and executing a sequence of activities (called *lifecycle*) will generate events and consequently change the entity's states (i.e., its attribute values). Detail explanations on how such activities consume model time (i.e., tracing model execution) and how model states are used to measure various system performance can be found in many textbooks (e.g., Banks, 1998; Harrell, Ghosh, & Bowden, 2004; Kelton et al., 2004; Law, 2007).

There are two paradigms to study the dynamic behaviour of a system. One focuses on transient entities' lifecycles called *material-driven*. Another one focuses on resident entities' lifecycles called *resource-driven*. Both paradigms have their own advantages and disadvantages in terms of execution speed and simulation output accuracy.

The material-driven paradigm is used for a system with few transient entities but with numerous resident entities. Since this system is examined based on the flow of transient entities (that their lifecycles are typically detail than resident entities), we can collect experiences of individual transient entities in much more detail. The advantage of this is that entities' animations and statistical output analysis can be more interesting. However, the increment number of transient entities will consume a lot of computer memory and consequently cause simulation execution becomes so slow.

The resource-driven paradigm is typically used for a large and highly congested system; i.e., a system that contains various transient entities demanding some services. This scenario could be found in a transportation system with many vehicles or a service system with many customers. Since there are relatively many transient entities compared to resident entities, it is more efficient to view model behaviour based on resident entities' lifecycles. The advantage of this paradigm is that since resident entities lifecycles typically involve few phases (e.g., idle or busy) and variables (e.g., their capacities, queue sizes, etc.), computer memory requirements and simulation execution speed are *insensitive* to system congestion caused by the increment number of transient entities. However, statistical outputs related to individual transient entities are limited since their lifecycles are not in focus in the model development. The material-driven paradigm is a better choice for animated DES models that focus more on entities' animations and state value visualizations.

3.3 Modelling Time

To sequence state transitions in DES, two dominant modelling styles (world views) are used: *event-oriented* and *process-oriented*. The choice of which modelling style should be used depends on a developer's familiarity with these concepts, their programming expertise (procedural or OOP) and time constraints.

Updating model time needs a component called a *monitor*. The *monitor* updates model time by jumping from event to event. During these processes of activating and cancelling events, various model statistical performances can be computed. The ideas of how model events are stored in an *Agenda* or an *Event List*

(i.e., a component for maintaining a list of events to be executed) make both approaches different.

3.3.1 The Event-Oriented Approach

The event-oriented (or event-scheduling) models a system's behaviour based on a set of events triggered by entities. Instead of grouping a series of events into a process description, it only lists events (no matter to which entity it belongs) based on their time of occurrence. Executing relevant event routines will simulate the system's processes and consequently update its model states.

This approach is well suite to model a system with a few types of entities since all relevant aspects of scheduling can be coded explicitly. This approach however becomes complicated and difficult to program when there are different types of transient and resident entities in a system (that introduce various kinds of events). Simulation tools that implement this approach include SIMAN (C. Dennis Pegden, 1989), SLAM (Pritsker et al., 1994) and SLX (Henriksen, 1997).

Figure 3.1 shows the execution mechanisms of the event-oriented approach. The *Event List* consists of a set of time-sorted event references (*Event ID*); each of which points to an event routine (*Event_1*, *Event_2*, etc.). At a particular point of time, the *Monitor* invokes the imminent event pointer in the *Event List* and activates its appropriate event routine. Executing a segment of code (*Descriptions*) for this event routine will schedule a new event that will later be inserted back to an appropriate location in the *Event List*. Consequently, the *Monitor* updates the *Simulation Clock*.

There are two options for advancing a model clock under this approach: *next-event time* and *fixed-increment time*. The next-event time advances model time to the most imminent future event time. At this point of time, the computer executes event routines, updates model states and determines the next scheduled event time. The advantage of this is that it saves computer time to run simulation since model time jumps from event to event. The fixed-increment time meanwhile advances model time to a fix amount of time unit. Model states (if one or more events have occurred) that have happened between these intervals will only be updated at the end of the intervals. The main downsides of this are: (1) the use of small time intervals but no

events occurred during the interval will only cause wasteful scanning and additionally impose computational costs, and (2) the use of big time steps but many events have occurred during the interval will suffer output accuracy since all state changes are only updated at the end of intervals.

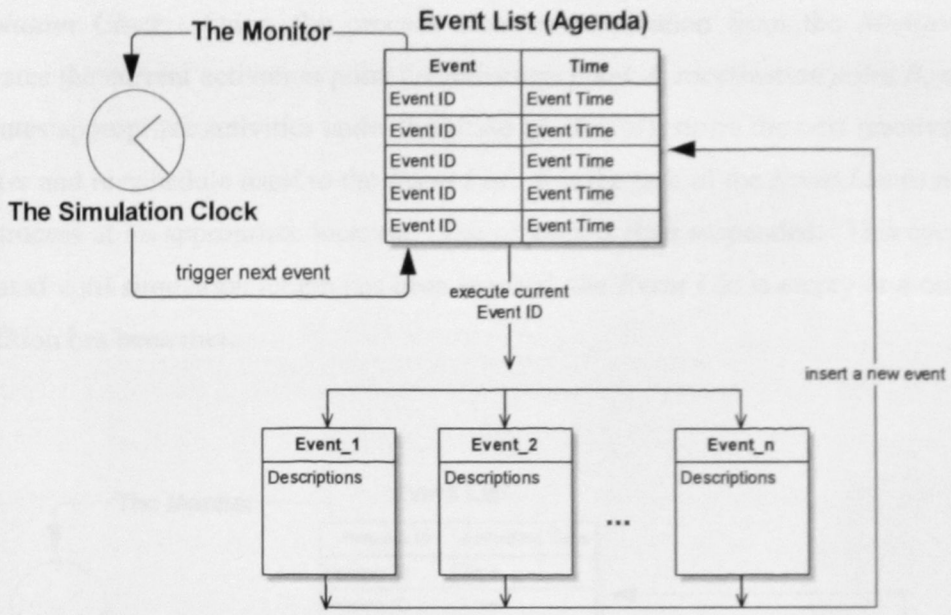


Figure 3.1 The Event-Oriented Approach Mechanism

3.3.2 The Process-Oriented Approach

The process-oriented approach is based on SIMULA (Birtwistle, 1980). It represents system behaviour from the point of view of active entities (called processes); each of which has its own lifecycle; i.e., a sequence of activities to be performed. Each process can either be in one of three phases: *active* (i.e., when its relevant activities are being executed), *passive* (i.e., when the process is suspended) or *death* (i.e., when the process has exhausted its actions). Only active phases (i.e., phases with time delays) update simulation time and model states.

A process can either be suspended for a *definite time* (delayed until a certain amount of time) or an *indefinite time* (delayed until some conditions are true; e.g., waiting to be re-activated by other processes). When a process is suspended, the *Monitor* retrieves the next imminent process from the *Event List* and then reactivates it. The process then flows itself to the next phase of its lifecycle.

Figure 3.2 illustrates the process-oriented mechanisms. Compared to the event-oriented approach that its *Event List* stores a set of time-sorted event routine pointers, the *Event List* in the process-oriented stores a time-ordered set of process identifications and their activation times (*Process ID, time*). At a particular point of time, the *Monitor* retrieves the imminent process from the *Event List* and updates its *Simulation Clock*. Once, the process receives notification from the *Monitor*, it activates the current activation point (*reactivation point A*, *reactivation point B*, etc.), executes appropriate activities under the phase (*Activities*), stores the next reactivation pointer and re-schedule itself to the *Event List*. It is the task of the *Event List* to insert the process at an appropriate location. The process is then suspended. This cycle is repeated until simulation length has been reached, the *Event List* is empty or a certain condition has been met.

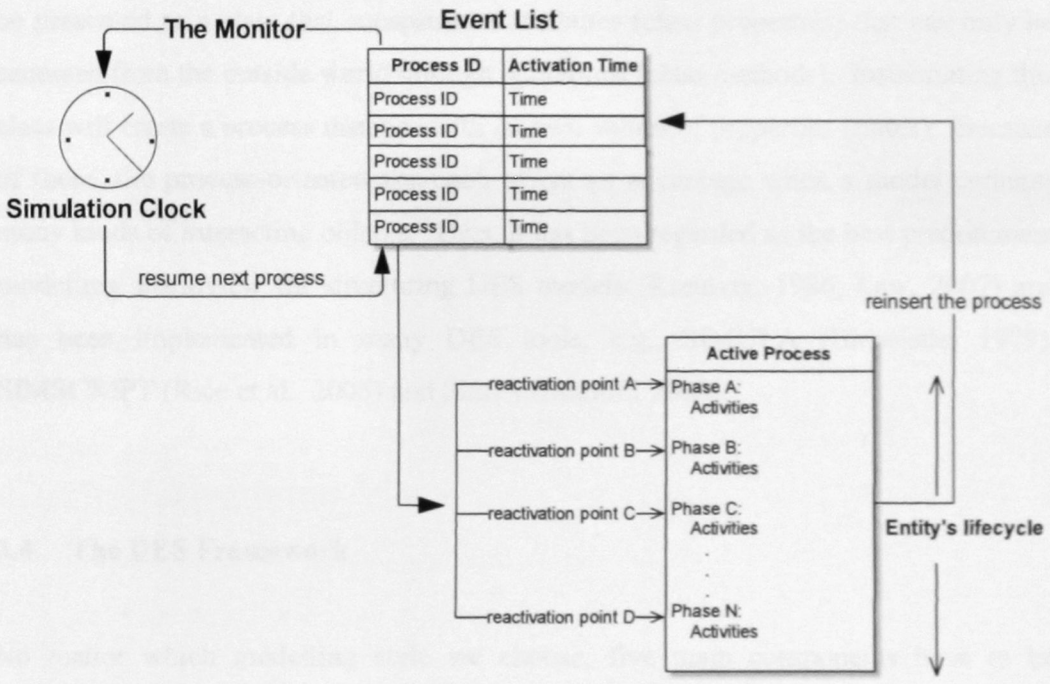


Figure 3.2 The Process-Oriented Approach Mechanism

The process-oriented approach is usually implemented using languages that support *co-routine* that allows multiple entry points for suspending and resuming execution at a certain location of a subroutine (e.g., C#, Python, etc.) or

multithreading that allows more than one activities to be performed in parallel within an application (e.g., Java, Ruby, etc.). However, any object-oriented languages can be used to implement this approach. Handling the process-oriented using object-orientation offers some benefits: (1) object-orientation is a natural framework for handling the complexity of the process-oriented framework through its concepts of *objects*, *classes*, *properties*, *methods* and *messages* thus easing the creation a class of entities, (2) object-orientation ensures that information is localized through the *encapsulation* concept thus simplifying the maintenance of entities' states and behaviour, and (3) object-orientation promises flexibility than conventional procedures by supporting *inheritance*, *polymorphism* and *composition* concepts thus easing the creation of various types of entities and their class maintenances.

The object-oriented approach eases the implementation of the process-oriented approach that views a system as a set of entities that interacts with each other to accomplish specific goals. In the object-oriented framework, a group of processes can be presented as a class that encapsulates attributes (class properties) that can only be accessed from the outside world through operations (class methods). Instantiating this class will create a process instance with its own values of properties (states). Because of these, the process-oriented approach offers an advantage when a model contains many kinds of interacting objects. Thus, it has been regarded as the best predominant modelling worldview for structuring DES models (Kreutzer, 1986; Law, 2007) and has been implemented in many DES tools; e.g., SIMULA (Birtwistle, 1979), SIMSCRIPT (Rice et al., 2005) and SimPy (Matloff, 2008).

3.4 The DES Framework

No matter which modelling style we choose, five main components have to be provided to structure and execute DES models: *entities* to represent objects, a simulation *clock* to manage current model time, *distributions* to generate entities' stochastic behaviour and drive model probability (i.e., for sampling model-time consuming activities), a *monitor* to manage interactions between entities, and *statistical instrumentation* to gather, analyze and report relevant aspects of simulation results.

Constructing these components should be based on an appropriate framework. This framework must be *transparent* to support extensibility (i.e., further extension to its base structures) and *well-designed* to avoid future amendments of its base structures. Thus, we constructed our own framework to serve as a base for the development of our DES tools. This design was based on the functionality of certain class libraries such as DEMOS (Birtwistle, 1979) and Psim-J (Garrido, 2001), and available frameworks such as SIMFONE (Rossetti, Aylor, Jacoby, Prorock, & White, 2000) and DESMO-J (Meyer, Page, Kreutzer, Knaak, & Lechler, 2005b).

We designed our own framework because of two reasons. First, most simulation textbooks and literature use available tools to build DES models. The tools' frameworks are hidden, making their reliability and extensibility to support our tool's objectives is restricted. Second, although some simulation textbooks that focus on simulation programming present their foundation frameworks (e.g., SIMFONE and DESMO-J), these frameworks (especially the entity and the Monitor classes) can only be implemented in languages that support co-routine or multi-threading (to continue and interrupt entities' lifecycles). Although this offers some advantages especially in allowing simulation to operate faster on computer systems that have multiple CPUs, they cannot serve as the base of the development of simulation libraries in any OOP programming languages. Thus, OOP languages that do not support co-routine and multi-threading (e.g., C++, ActionScript, etc.) cannot implement the frameworks. Our framework is divided into four packages based on their functionality:

- Data Collectors
- Distributions
- Monitor (Simulation Executive)
- Resource (Servers and Queues)

Figure 3.3 shows a package diagram that depicts the dependencies between these packages in order to create queuing network models. Note that this framework has been presented in Khalid, Kreutzer and Bell (2009).

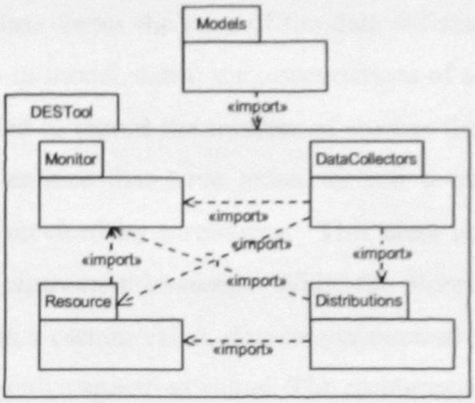


Figure 3.3 Package Diagram for Queuing Models

3.4.1 The *Data Collectors* Package

Facilities for statistical instrumentation and reporting are essential features in DES models. Thus, to gather, analyze and report statistical information generated during simulation runs, the *Data Collectors* package must be available. This package should consist of seven classes: *Collector*, *Counter*, *Tally*, *Histogram*, *Boxplot*, *Graph* and *GraphEntry* (see Figure 3.4).

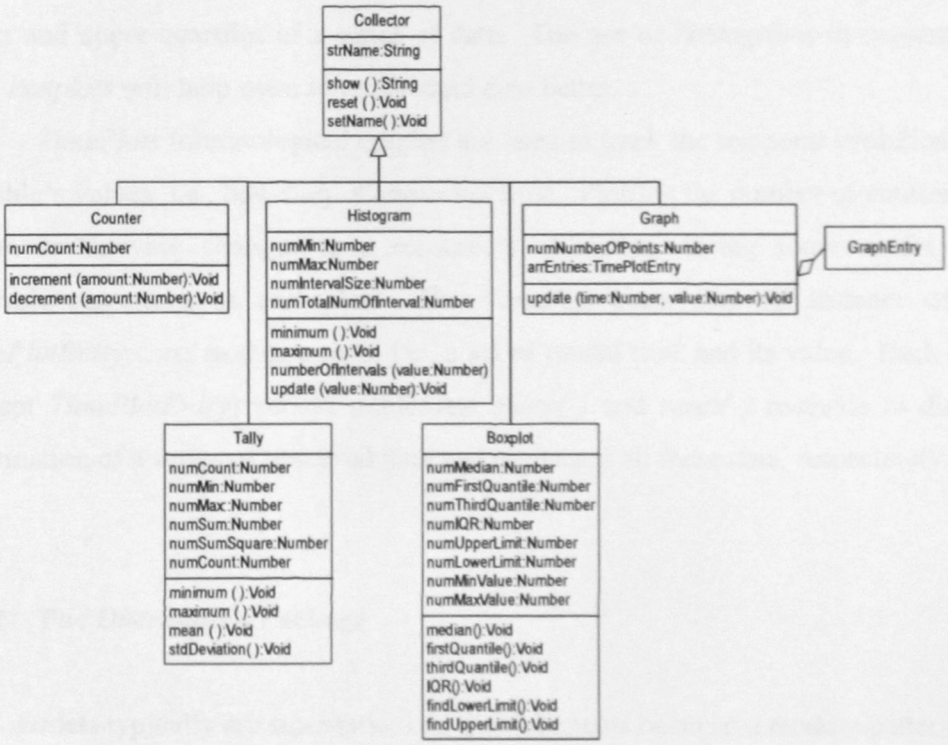


Figure 3.4 Class Diagram for the *DataCollectors* Package

The *Collector* class forms the base of the data collector hierarchy. *Counters* record relevant changes in model states; e.g., occurrences of significant events. They can, for example, be used to record the number of entities that have entered or left a model, the number of entities that have joined or left a queue, or the number of entities that have been serviced by a resource. This class consists of two methods: *increment(amount)* and *decrement(amount)*. While the *increment(amount)* is used to increase the counter with a certain value, *decrement(amount)* should also be provided to decrease the counter with a specified value. The combination of the two methods is always used in an object; e.g., to report the number of entities in a queue object or in a resource object. Note that we have to provide flexibility for users to specify the *amount* number in case they want to represent a *batch* arrival or departure.

A *Tally* reports the minimum, maximum, mean and standard deviation of a series of values. It can, for example, be used to gather reports on delays; e.g., time spent waiting in queues or residence times in the model. *Histograms* assign values to intervals and show frequency counts for each interval in graphical forms (bar charts). They can be used to gather and report, for examples, time between arrival of entities, time waiting in a queue, service times of a resource and cycle times. *Boxplots* provide descriptive statistics of data variation. They can be used to graphically report information about the smallest, largest and median values of observations, and the lower and upper quartiles of a series of data. The use of *Histograms* in conjunction with *Boxplots* will help users to understand data better.

TimePlots (chronological graphs) are used to track the temporal evolution of a variable's values; i.e., how they change over time. Plotting the number of entities in a queue or showing changes to a resource's utilization during some model time intervals can serve as examples. The *TimePlot* class uses an instance of the *TimePlotEntry* class as data points; i.e., a set of model time and its value. Each class (except *TimePlotEntry*) should implement *show( )* and *reset( )* methods to display information of a series of observed data and to discard all these data, respectively.

3.4.2 The *Distribution* Package

DES models typically are stochastic; i.e., their elements occur in a random pattern that eventually generates random events. For example, each entity has its own arrival time

and travel times (from location to location) that will generate non-deterministic results. Experimentations with these inputs to find the best possible outputs in various scenarios are one of the purposes of DES. Simulating this random behaviour requires a component that has capabilities for generating samples from a variety of distributions.

The *Distributions* package provides a selection of pre-packaged distribution objects. These may, for example, be used to schedule the time between workload items' arrivals or service times of resources. Note that the term "RNG", used in Figure 3.5, stands for *random number generator*. There are two methods to generate computer random numbers: the *middle square* method (Knuth, 1981) and the *congruental* method (Boyar, 1989; Hull & Dobell, 1962). The main limitations of the first method are the iterations for generating new random numbers cannot be longer than 10^n, where n is the number of digit random numbers and if the first half digits of generated numbers are zeros, the subsequent numbers will then be decreasing to zero and this will eventually stuck the generator. The advantages of the second method are that (1) this method is easy to understand and be implemented in addition to producing decent random numbers with the right choice of its coefficients, and (2) this method only needs minimal computer memory to retain its state.

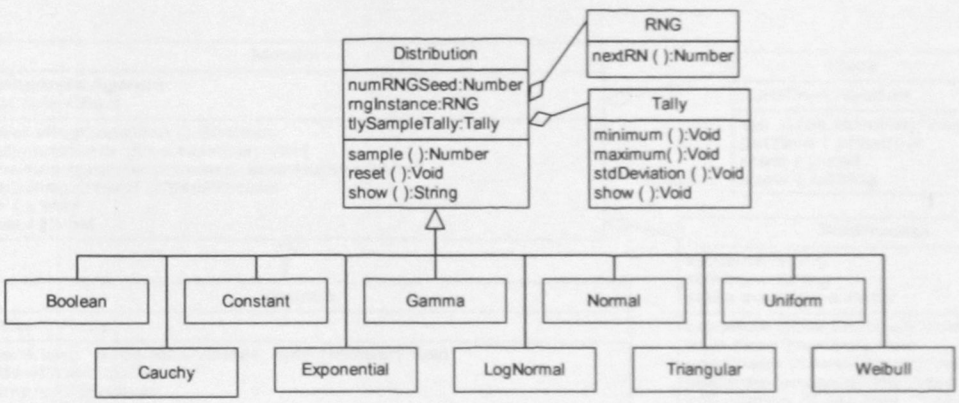


Figure 3.5 Class Diagram for the *Distribution* Package

We use Actionscript's generator, which is based on the standard congruential method, for this purpose. The *nextRN( )* method is used to create random numbers uniformly distributed between 0 and 1, which are then used in distribution functions. Examples are *Boolean*, *Exponential*, *Gamma*, etc.; each of which represents a

statistical analysis of empirical data either collected from a real system or an approximation of sample data for an imagination system. More comprehensive discussion on estimating an input distribution and its characteristics can be found in any textbooks; e.g., by Banks (1998) and Law (2007).

Each distribution class has a *sample( )* method that implements a function of a random number for generating distribution samples. These samples can be updated in a *Tally* instance (through a composition technique) to report basic information (e.g., the minimum, maximum, etc.) of a series of generated data. Options to show and remove these data should be available through *show( )* and *reset( )* methods.

3.4.3 The *Monitor* (Simulation Executive) Package

The *Monitor* package provides the infrastructure for sequencing state transitions in DES models. Its main focus is on the creation, scheduling and termination of processes. This package consists of five classes: *SimProcess*, *Monitor*, *Agenda*, *Clock*, and *Event* as shown in Figure 3.6. The *SimProcess* class describes the life cycles (i.e. the sequence of events such an entity moves through) of active entities.

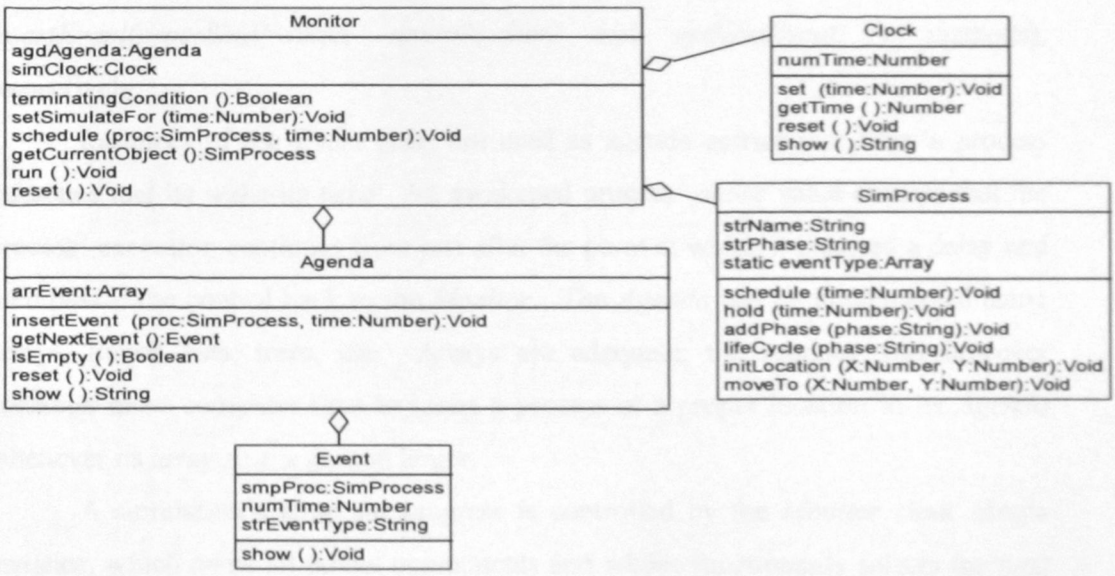


Figure 3.6 Class Diagram for the *Monitor* Package

Two important methods should be provided in the *SimProcess* class: *schedule(time)* and *hold(time)*. The *schedule(time)* method is to schedule an instance of *SimProcess* with a specific time value. The *hold(time)* method is to delay this process until a specific value of future time (i.e., current model time plus a specified amount of delay time). When the time is reached, this process will be activated so that it can flow to the next phase of its lifecycle and once again one of the two methods will be called again until it is destroyed. Since Actionscript 2 does not offer any features for implementing *co-routines* or *threads*, each *SimProcess* instance needs to keep track of its current *phase* (i.e., the current stage of its lifecycle) using a class variable. This property is updated whenever the process encounters a model time delay. Tracking *SimProcess* instances' current phases needs the *SimProcess* class to compose a *Monitor* instance so that they can insert themselves to the *Monitor*'s *Agenda*.

The *Monitor* owns an *Agenda* (or known as an *Event List*) that maintains a time-ordered list of future events. Whenever a new event is scheduled, the *Monitor* inserts a process and its time reference (*event notice*) at an appropriate agenda position and will then wake and remove this process whenever its time of occurrence is reached. Thus, the *Monitor* should have two encapsulated methods; i.e., *schedule(proc:SimProcess, time:Number)* and *getCurrentObjects( )* to insert and remove processes from the *Agenda* (by delegating tasks to the *Agenda*'s *insertEvent(proc:SimProcess, time:Number)* and *getNextEvent( )* methods), respectively.

Instances of the *Event* class are used as agenda entries that store a process reference and its wake-up time. An awakened process' *phase* value ensures that the process' execution continues from just after the point at which it incurred a delay and then passes the control back to the *Monitor*. The *Agenda* can be implemented using arrays, linked lists, trees, etc. Arrays are adequate; the *Monitor* will however consume more computer time to insert a process at a proper location in its *Agenda* whenever its array size is getting larger.

A simulation's temporal progress is controlled by the *Monitor* class' single instance, which owns all model components and whose functionality selects the next imminent event from an agenda, updates the model clock (an instance of a *Clock* class) to the relevant time value, and activates the appropriate process, instructing it to execute its next phase. This executing process is repeated until the *Agenda* is empty

(*isEmpty( )*), a certain condition has been met (*terminatingCondition( )*) or simulation time has been reached (*setSimulateFor(time)*). Thus, to avoid an empty *Agenda* for the first run, it is important to ensure that at least one process has been placed in the *Agenda*. Executing this process will transfer it to other phases and/or create a new process.

3.4.4 The *Resource* (Servers and Queues) Package

Figure 3.7 shows a class diagram for the *Resource* package. This package consists of two classes: *Server* and *Queue*. Both the *Server* and *Queue* classes can compose instances of *Tally*, *Graph*, *Histogram* and *Boxplot* to report their states in various formats.

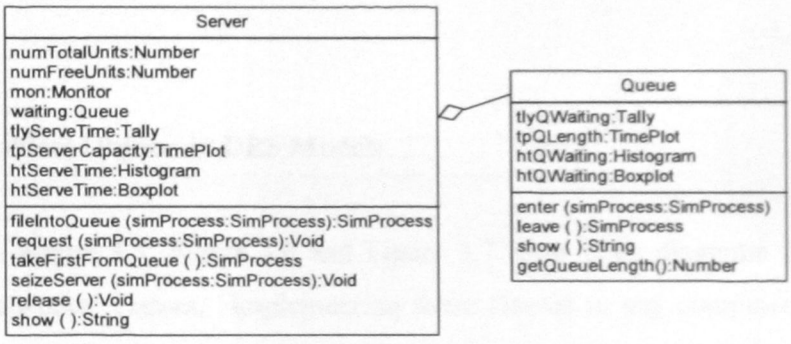


Figure 3.7 Class Diagram for the *Resource* Package

Servers allocate limited capacity resources to service requests. If a server's capacity is exhausted, the requesting entity will be placed in a *service queue* - an instance of the *Queue* class. As the *SimProcess* class, the *Server* class must compose a *Monitor* instance so that its lifecycle can be tracked.

The *Queue* class should implement two methods: *enter(simProcess)* and *leave( )*. The *enter(simProcess)* method is to insert a *SimProcess* instance to a queue while the *leave( )* method is to retrieve the head of the queue. These two methods are used in the *Server* class through a composition technique. Among methods that should be provided for the *Server* class include:

- *fileIntoQueue(simProcess)* is to insert a *SimProcess* instance into a queue before allocating the instance with a certain unit of the server. This method uses the *Queue*'s *enter(simProcess)* method to accomplish this task.
- *request(simProcess)* is to check if the *Server* is ready to allocate its service; i.e., if it can supply a certain amount of unit for a requested *SimProcess* instance.
- *takeFirstFromQueue( )* is to enable the *Server* to retrieve the first *SimProcess* instance from a queue . It calls the *Queue*'s *leave( )* method to accomplish this task.
- *seizeServer(simProcess)* is to allocate a certain unit of the *Server* 's capacity to a requested *SimProcess* instance.
- *release( )* is to enable the *Server* to get back a certain amount of unit that it has allocated to a *SimProcess* instance, so that the next *SimProcess* instance can request for its service. Once again, the *request(simProcess)* method will be called.

3.5 Graphical Objects in DES Models

Figure 3.4, Figure 3.5, Figure 3.6 and Figure 3.7 show class diagrams for creating queuing networks' classes. Implementing these classes in any computer languages eases model building through API. The resulting models are however limited to text description models; i.e., a list of texts that describes their logic and behaviour. Creating graphical structures and animated versions of the models needs the concept of graphical objects that symbolize their functionalities and ease access to model properties.

Graphical objects for animating DES models can be split into two different categories. The first one is independent of the simulation domain or *Domain Independent Objects*, while the second one is specific to a particular type of simulation or *Domain Dependant Objects*; see Figure 3.8.

Domain Independent Objects can be further divided into two subgroups: *static* objects and *dynamic* objects. *Static* objects do not move or change visual appearances during animation; e.g., simulation inputs (i.e., different types of distributions under the *Distribution* package) or symbols for the simulation controller (i.e., the *Monitor).*

Dynamic graphical objects, on the other hand, change their appearances and/or locations. This category includes clocks (under the *Monitor* package), histograms, graphs and boxplots (under the *Collector* package) and queues (under the *Resource* package).

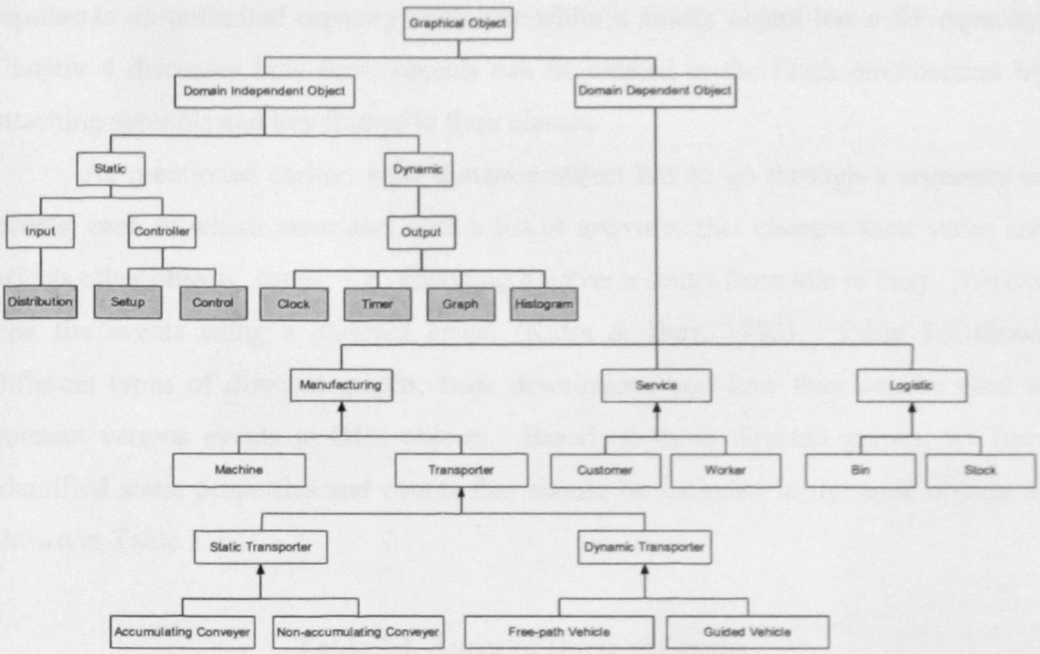


Figure 3.8 *Graphical Objects* in DES

Domain Dependent Objects are often dynamic objects that represent *SimProcesses*' changing location (e.g., moving customers or vehicles) and/or appearance (e.g., machines or conveyor belts). Figure 3.8 depicts some examples of domain dependant objects for service, manufacturing and logistic systems. In manufacturing systems, transporters are used for transporting entities from location to location based on a mean velocity value. Transporters are of two types: *static* (conveyers) and *dynamic* (vehicles). While vehicles move along with entities, conveyers remain at the same places; i.e., they only move entities from location to location using belts based on the velocity of the belts.

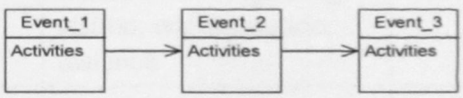
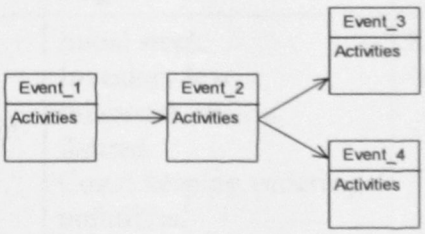
As shown in Figure 3.8, there are two types of vehicles: *free-path* and *guided*. Free-path vehicles can move freely between stations and are not influenced by other transporters' traffic. Examples are trucks, forklifts, etc. Guided vehicles (e.g., automated guided vehicles) run on fixed networks (tracks or rails) and are influenced

by traffic congestion. Conveyers meanwhile are of two types: *accumulating* and *non-accumulating*. Accumulating conveyers will keep moving although they have been accessed by entities. On the other hand, non-accumulating conveyers will stop their belts for loading or unloading entities.

In logistic systems, *bins* and *stocks* are used for holding goods. A bin object represents an unlimited capacity container while a stocks object has a fix capacity. Chapter 4 discusses how these objects can be created in the Flash environment by attaching symbols and key frames to their classes.

As mentioned earlier, each dynamic object has to go through a sequence of events; each of which associates with a list of activities that changes their states and affects other objects' states; e.g., changing a server's status from idle to busy. We can link the events using a *directed graph* (Kalra & Barr, 1992). Table 3.1 shows different types of directed graphs, their descriptions and how they can be used to connect various events in DES objects. Based on these directed graphs, we have identified some properties and events that should be included in dynamic objects as shown in Table 3.2.

Table 3.1 Types of Directed Graphs

Directed Graphs	Descriptions	Examples
Time line	A linear arrangement of events. Each object must follow a fix sequence of events; i.e., one event will only lead to one other event. 	Entities with a fix path.
Time tree	A few alternatives of events. An event can traverse to several possibilities of the next events. 	Entities with a diverse sequence of events; e.g., a model that considers *decision points*, *balking* (arriving entities that do not join a queue but go away), *reneging* (entities that join a queue at first but decide to leave the queue later) or *jockeying* (switching queues).

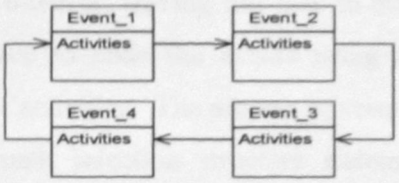
Time graph	A loop of events; i.e., a series of events that is repeated by an object.  	Servers, transporters conveyers, bins and stocks.

Table 3.2 Properties and Events for Dynamic Objects

Objects	Properties	Events/Phases
Entity	Initial location Current location Target location Arrival time Departure time	Arrive, Depart and events associated with other communicated objects
Server	Capacity Service Time Status: idle or busy Utilization	Request, Seize, Delay (Busy), Release (Idle), Inactive and Fail
Transporter	Status: idle, busy or inactive Velocity Time unit Capacity Current load Initial position Distance set: beginning station, ending station, distance	Request, Load, Transport, Free and Stop
Conveyer	Velocity Units Cell size Segment: beginning station, next station, length	Access, Convey, Exit and Halt
Stock and Bin	Initial stock Inventory levels: minimum, current, desired Costs: keeping, ordering, unfulfilled	Request, Product Delivery and Stock Order

Time graph entities can be hard coded by tool designers while *time tree* entities that traverse to several paths of events must flexibly be coded by model developers. However, leaving this task in their hand could create certain problems. First, they have to code the events using *if-else* or *switch-case* statements with descriptions of activities. The process of creating, extending and saving entity classes and writing such selection structure statements may burden and cause tension. Second, at certain levels of *if-else* or *switch-case* statements, they again have to write another selection structures so that at the relevant stage certain entities can skip linear events to represent an alternative flow; e.g., based on certain probability, queue length, work in process, etc. These processes tend to make code clumsier and lead to logical errors. This problem is getting worse if there are many classes of entities in a model, each of which has their own alternative paths. Third, they have to carefully study a segment of relevant code if they plan to modify entities' lifecycles to ensure that the modification will flow the entities along the right paths. We have catered these problems by generating events during runtime instead of specifying events during design time. This approach will be discussed in details in Chapter 5.

CHAPTER 4

USING FLASH FOR SIMULATION

4.1 Introduction

The use of simulations in education and training is an attractive idea since it allows learners to gain access to and experiment with dynamic models under different scenarios. However, to take full advantages of the technology's potential, simulations must be interactive enough to allow learners to fully immerse themselves rather than tediously studying lists of results or just watching pre-recorded animations of simulation experiments.

Visualizing DES models in an *attractive and interactive* environment is suspected to help learners to learn and understand DES systems better. While most DES tools offer some capabilities to generate animations, simulators with a strong feature set for animation design typically stress qualitative understanding of system behaviour rather than statistically well corroborated predictions of system performance. Thus, supplying teachers with easy-to-use tools (e.g., through a drag and drop approach) that create highly animated models to motivate learners, equipping the models with dynamic displays and means of interactions to engage learners and easing the deployment of the models either on the web or modern LMSs to serve communities of learners are crucial. Unfortunately, no single current DES tools have been fashioned for these.

Attractive and interactive DES models integrate simulations and animations to reflect *change* in either the time or space dimension. *Temporal* change, for example, occurs whenever a simulation encounters delays (in model time) and whenever an animated object changes appearance. *Spatial* change occurs whenever a visual entity *moves*. To support *animated simulations* requires a nested design, where model time must be mapped onto animation time, and animation time must be mapped onto real time. There are a number of strategies for connecting such layers of representation.

We have however opted for a *concurrent (synchronous)* approach, where model time is always *proportional* to animation time and animation time is always *proportional* to real time.

The design of DES tools should be based on Visual Interactive Simulation (VIS) fundamental concepts. For this reason, we start this chapter with a brief review of the concepts and the benefits they offer to learning and teaching. Some available approaches for integrating simulation and animation in VIS models will also be introduced. Based on the concepts and a selected integration approach, we then argue that Adobe Flash is a suitable development environment for constructing tools of VIS models. A proposal of how VIS's essential components can be created with the help of our framework (discussed in Chapter 3); i.e., how we relate all the four packages to a single overall class diagram for VIS models is then presented. We then present a series of simulation components that have been developed to build queuing models. We further our discussion by listing some tricky issues in integrating an animated simulator to DES models specifically in permitting animation speed to be dynamically adjusted during runtime. This chapter ends with a presentation of an overall class diagram that supports DES for logistic and manufacturing systems.

4.2 Visual Simulation and Visual Interactive Simulation

Interactive simulations use tools that focus on either model *developers* (e.g., teachers) or *consumers* (e.g., learners). The first type of tool helps developers to specify model structures and model parameters within a *graphical programming* environment; e.g., through blocks and symbols, or by answering a series of questions. The second type of tool uses *animation and interaction* for showing a model's behaviour either during or after a simulation run.

Model building through blocks and symbols typically gives developers more flexibility in constructing models than answering a series of questions that constrains developers in only choosing models from a set of pre-fabricated models, considered by the mindset of tool designers. Since both approaches focus on *building* a model using some means of interactions, it is well-known as *Visual Interactive Modelling* (VIM). Au & Paul (1996), Odhabi, Paul, & Macredie (1998) and Sargent (2004) discuss such simulation software.

The resulting models can be of two types: *Visual Interactive Simulation* (VIS) or *Visual Simulation*. While simulation models that permit users to interact with them during their execution are referred to as Visual Interactive Simulation, any simulation model that only allows users to view its behaviour through animations without any capability for interaction is known as Visual Simulation (see Bell, 1989; Mascarenhas, Rego, & Sang, 1995; S. Narayanan et al., 1997; S. L. Robinson, 1994; Sargent, 2004). Visual Simulation focuses on the *attractiveness* of simulation by tracing and surfacing the dynamic behaviour of models through graphical forms. They typically support two types of graphic displays: *abstract displays* and *representative displays*; see Rooks (1991) and Figure 4.1.

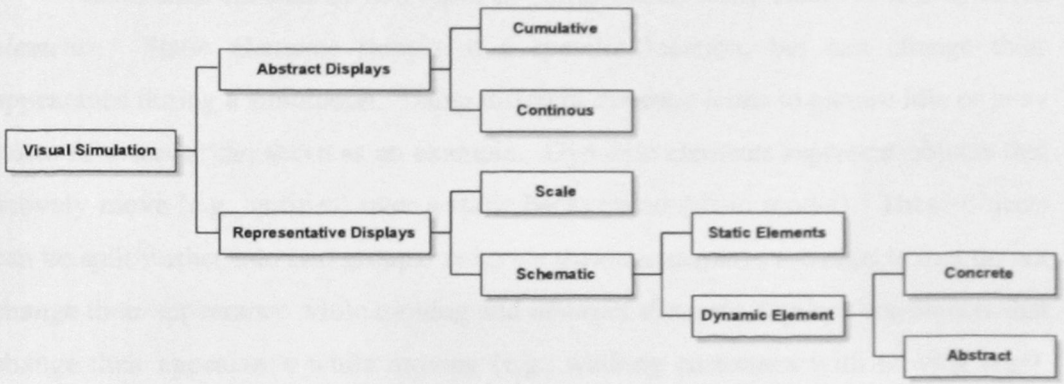


Figure 4.1 Visual Simulation Components

Abstract displays stress on *data visualization* of model states. They are used for interpreting and enhancing the presentation of statistical data (e.g., the *Data Collector* package in Chapter 3) in the simplest form that can be comprehended by consumers. Various visualization methods (e.g., the use of colour, appropriate texts, etc.) that engage them and promote their understanding could be implemented. Abstract displays can be further divided into two groups; i.e. *cumulative* and *instantaneous* displays. As the name suggests, cumulative displays increase the amount of data shown during a simulation's execution. *Past* data points will remain on display until removed by model developers or consumers. Cumulative displays help document the values of model *variables' change* over time; e.g., the number of entities in a queue. Examples are graphs, progress bars and scatter plots. Instantaneous displays, on the other hand, only expose *current* states of model

variables during a simulation run, without showing their previous states. Examples include histograms, bar charts, pie charts, gauges that indicate levels, etc.

Representative displays offer *pictorial* views of a model in a simplified form. They can be of two types: a *scale model* or *schematic*. A scale model gives a pictorial view of a system drawn prior to starting a simulation and will not change during a simulation experiment. It typically offers the physical layout of a model, trying to offer a realistic background in front of which the simulation takes place. Schematic displays are more abstract. They are used to visualize the topology and paths of movement within a simulation and are typically required for *animations*. While a scale model is completely static, schematic displays serve to frame changes during a simulation run.

Schematic consists of two types of components: *static elements* and *dynamic elements*. Static elements remain at a specific location, but can change their appearance during a simulation. Using different *dynamic icons* to picture idle or busy states of a server can serve as an example. Dynamic elements represent objects that actively move (e.g., entities) over a static background (scale model). These objects can be split further into two groups: *concrete dynamic displays* are objects that do not change their appearance while moving and *abstract dynamic displays* are objects that change their appearance while moving (e.g., walking customers with moving legs). Henriksen (2000) further differentiates these objects based on their types of motion; i.e., objects that only move in a linear form between two fixed points (*absolute* movement), or objects that move along defined paths (*guided* movement); see (Kamat & Martinez, 2007).

Animations create and change the appearance of images at different points in time to convey visual information to viewers. In DES, animations are used mainly to observe patterns of movement of entities including their transformation from one state to another, their interactions with other objects, and the occurrence of queues whenever capacity-constrained resources cannot be seized. To attain advantages over traditional DES models, some researchers (e.g., Belfore et al., 2003; Gilman, 1985; Hill, 1996; Macal, 2001; Rekapalli & Martinez, 2007; Stahl, 2003; Wenzel & Jessen, 2001) suggest a few alternatives. This includes presenting a model in a more user-friendly and understood form (e.g., model developers should clearly illustrate model structures with appropriate symbols and label names on a stage, and display simulation results in a graphical form with appropriate headings, labels, etc.),

providing graphical interaction windows for receiving input from their users (model consumers) and designing multiple windows to display simulation information so that users can customize their views of the model.

Simulation, animation and interaction are core components for VIS. Basically, VIS models allow learners to (1) initialize simulation parameters and run the model, (2) observe the simulation behaviour through animation, (3) experiment by making changes to model parameters while a model is running, (4) re-observe the impact of the changes, and (5) customize model visualization during a model execution. Since the very notion of *simulation* implies experimentation with models (Rooks, 1991), such runtime interaction capabilities should be an integral part of any advanced computer-based simulation development tools. Providing the interaction requires us to examine some DES animation approaches.

4.3 Animation Approaches

Dynamic elements focus on object movement from location to location, satisfying their time delays. For this, relevant information from simulation needs to be mapped with animation. This mapping process can be based on three available approaches; i.e. *post-processed animation, direct simulation-animation* and *concurrent animation* (see Figure 4.2).

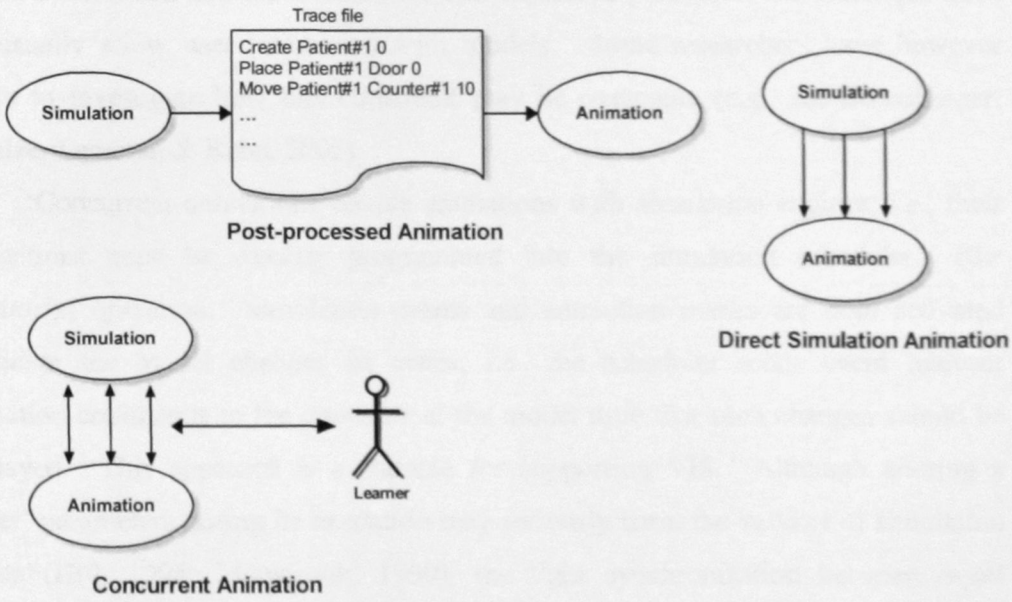


Figure 4.2 Three Approaches to Combine Simulation with Animation

Post-processed animations separate simulation and its animation. An animation is performed after a simulation has been run to completion (Hill, 1996; Rohrer, 2000). To drive an animated model, an animation tool needs to read a simulation *trace file* that contains relevant data as well as time-ordered command sequences; e.g., commands for creating, moving or destroying entities. Trace files can be written using a simulation package that provides the capability of writing to text files during a simulation run, general purpose programming tools or a text editor. Because of their reliance on pre-collected data, post-processed animations cannot support any runtime interaction between users and a simulation model. Despite this drawback, they offer some advantages such as (1) animation tools and simulation tools can be independent in terms of software and operating systems, (2) no computer memory is shared between simulation and animation tools that causes their executions become efficient, and (3) animation viewers can still jump backward and forward in the model time dimension and speed or slow down the rate at which sequences of events are displayed since all relevant simulation data has been collected.

Direct simulation-animation is a form of real time animation, in which a trace of simulation events and their visual displays are created on the fly; i.e., *during* a simulation run. Animation tools that support this approach must be based on some means that allow interaction with the simulation software at execution time; e.g., a Dynamic Link Library (DLL) in case of the *Proof* (Henriksen, 2000) software. Since the simulation and animation tools are still separated processes, the technique does not usually allow user interaction with models. Some researchers have however begun to investigate how this constraint may be overcome (e.g., see Strassburger, Schulze, Lemessi, & Rehn, 2005).

Concurrent animations couple animations with simulation engines; i.e., their interactions must be directly programmed into the simulation scheduler's (the *Monitor*'s) operation. Simulation events and animation events are *both* activated whenever the model changes its states; i.e., the scheduler sends event relevant animation commands to the *animator* at the model time that such changes should be displayed. This approach is a suitable for supporting VIS. Although altering a model' parameters during its execution may seriously harm the validity of simulation results (Hill, 1996; Matwiczak, 1990), the tight synchronization between event scheduler and animator permits flexible patterns of interaction with running models;

an often essential element for enhanced understanding of complex systems in training and education (S. Narayanan et al., 1997) and making the distribution of the models on the web or LMSs much easier. However, the proper connection between *simulation (model) time* (i.e., a set of important points of time (events) abstracted from a continuous process system where model behaviour and state changes take place) and *animation time* (i.e., a set of interval time to animate and move entities) is a challenge for developing the kind of tool.

Table 4.1 shows some aspects of simulation and animation approaches. Table 4.2 meanwhile lists interaction characteristics of concurrent and post-processed animations. Based on these characteristics, we have categorised some DES tools as in Table 4.3. As we can see, most of the tools are based on a *unidirectional* characteristic; i.e., their resulting models do not support runtime interactions and the models cannot also be executed on web pages. DES tools that are concurrent, bidirectional, homogeneous and integrated are important for building models for learning purposes.

Table 4.1 Aspects of Simulation-Animation Approaches

Aspect	Feature	
Mapping Approach	*Concurrent*: Animations are directly coupled with a simulation engine	*Direct, Post-processed*: Animation is performed after the entire model has been processed
Interaction	*Bidirectional*: Simulation and animation can react to each other	*Unidirectional*: Simulation controls animation
Hardware Platform	*Homogeneous*: Simulation and animation are executed on the same platform	*Distributed*: Simulation and animation can be executed on different platforms
Animation	*Integrated*: Animation is integrated in a simulation engine	*External*: Animation and simulation are independent

Table 4.2 Interaction Characteristics of Concurrent and Post-processed Animations

Interaction Characteristic	Concurrent	Post-processed
Ability to change simulation parameters and directly observe simulation results	Yes	No
Animation performance (speed, smooth motion)	Variable	Excellent
Ability to fast forward	Yes	Yes
Ability to rewind	No	Yes
Ability to run large models	Variable	Excellent

Table 4.3 Available DES Tools and Their Features

Simulation Tool	Feature
Proof	Concurrent/Direct, unidirectional, homogeneous/distributed, external
SLAM	Concurrent/Post-processed, unidirectional, homogeneous/distributed, integrated
Arena, AutoMOD, ProModel, Simul8, Extend, GPSS	Concurrent, unidirectional, homogeneous, integrated
SIMAN/CINEMA, SEEWHY/WITNESS, SLAM/TESS	Concurrent, unidirectional, homogeneous, external

4.4 Managing Simulation and Animation

Animated DES deals with animation of various entities in a system. Each entity is animated independently in terms of its dynamic appearance (transformation of physical displays from state to state), motion (movement from location to location) and interactions with other objects at appropriate instances of time; see Figure 4.3. The motion of DES' entities only employs *descriptive motion* (i.e., motion without considering factors that cause it) and *behavioural motion* (i.e., reactions of the object based on its communications with its environment during temporal interval) rather than *generative motion* (i.e., motion caused by some external factors; e.g., forces or torques that effect objects' position and orientation); see Donakian and Cozot (1995).

Linking a simulation model to its animation requires a conversion of three types of simulation information; see Table 4.4. The time difference between two consecutive events (see Table 4.5) and the resulting delay (in a model time unit) are the only information available for an *animator* to display changes of simulation entities' activities, location or appearance; e.g., to show a smooth glide between screen coordinates or changing an icon representing a server's idle state to one showing that it's now busy. Thus, anything happening between two consecutive events is considered irrelevant (i.e., outside the brief of the model) and therefore ignored.

<table>
<tr><td rowspan="5">Animated entity
  </td><td colspan="3">Visual physical dynamic appearance in 2D (images, geometries) or 3D (geometries) formats</td></tr>
<tr><td colspan="2">Properties with temporal states (values of properties) that change during simulation to adapt the current situation. Properties can be scalars (e.g., the current location, a transformation value, a velocity value, etc.) or vectors (the direction of movement)</td><td rowspan="3">Interfaces</td></tr>
<tr><td rowspan="2">Activities
(functions/operations)</td><td>Animation methods to define actions in response to events; e.g., creation, movement, translation, rotation, modification, communication, elimination, etc.</td></tr>
<tr><td>Event handlers to support runtime interactions with users; e.g., onClick, onMouseOver, etc.</td></tr>
<tr><td colspan="3">Events that modify entities' behaviour (internal states)</td></tr>
</table>

Figure 4.3 DES's Animated Objects

Table 4.4 Simulation to Animation Conversion

Simulation	Animation
Delay (time)	Continuous movement between two locations (time and space)
Events (state changes)	Visual appearance of objects' behaviour
Numerical output that is typically difficult to understand by learners	Visual format reports to ease learners' understanding

Table 4.5 Events and Model Time Difference in a Sample System

Time	Process	Event		Model Time Difference
0	Customer 1	Arrival		2
2	Customer 2	Arrival	Delay time for Customer 1	1
3	Customer 1	Seize Teller		2
5	Customer 2	Join Queue		3
8	Customer 1	Release Teller	Delay time for Customer 2	1
9	Customer 3	Arrival		0
9	Customer 2	Seize Teller		

Consistent transformations of model time to animation time (see Figure 4.4) are essential for maintaining the realistic illusion of a real system either its model is consistently running at a default rate or variably running at a user-specified rate. However, animated models that allow users to flexibility adjust their execution speed (i.e., to speed up, slow down or halt their model time) at any time they wish need to embed a term called a *viewing ratio*. A viewing ratio is used to map the given number of model time units into a corresponding number of seconds of animation time. For example, if the viewing ratio is set to 10, then 1 second of animation time is equal to 10 units of simulation time.

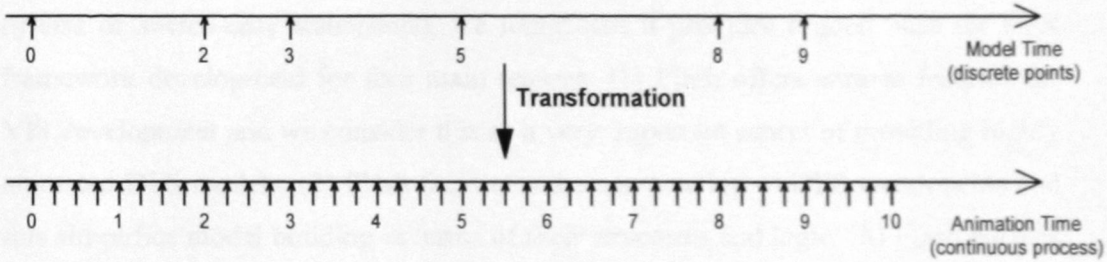


Figure 4.4 Transformation from *Model* to *Animation* Time

Equation 4.1 can be used to smoothly animate all transactions between events. This equation ensures that all state changes will be visible at their proper time, no matter what viewing ratio has been selected by users.

Animation time = *model time difference between two consecutive events * (1 / viewing ratio)* (Eq. 4.1)

Equation 4.2 can meanwhile be used to show smooth movement of an object from location to location. It ensures that the object arrives at its target location at a specified point of time, with a condition that a viewing ratio is smaller or equal to model delay. If the viewing ratio is greater than a certain entity's delay time, we need to set the movement to 1 to make sure that the object will arrive at one second animation time.

$$\textit{Movement (per unit animation time)} = \textit{distance} * \textit{(viewing ratio / delay to location)} \quad \text{(Eq. 4.2)}$$

4.5 Flash as an Implementation Language for Simulation and Animation

Adobe Flash (H. M. Deitel, Deitel, & Goldberg, 2004; Lopez, 2006; Mohler, 2006; Shupe & Hoekman, 2006) offers a tool for creating attractive, interactive and multimedia affect models. However, we have not found any reports on Flash-based DES models or Flash libraries for DES model construction.

We have therefore investigated Flash's features for its suitability as a DES development tool. In spite of the fact that Flash does not support *coroutine* that requires us to write the lifecycle of each type of active entity using selection structures (*if-else* or *switch-case* statements), we found that it provides a good base for DES framework development for four main reasons: (1) Flash offers various features for VIS development and we consider this as a very important aspect of providing highly animated DES models, (2) Flash facilitates the construction of DES components and this simplifies model building in terms of their structures and logic, (3) Flash enables model developers to locate animated objects on a relevant layer of multiple layers and this eases the management of various objects and GUIs, and (4) Flash automatically creates web-based models and supports web interactions and these ease model distribution. Additionally, its scripting language *ActionScript* is syntactically similar to Java and C++ in many ways; e.g., object-oriented structure, package, class, method, properties, data types etc. Thus, anyone who knows the languages and has some background in DES frameworks could easily implement the frameworks using Flash. Note that other tools exist or may appear that meet these criteria. However, at the time the research was done, Flash was a widely used tool that met these criteria. A

recent candidate would also be HTML5, although this is nowhere near as mature as Flash. It does have the advantage of working on Apple mobile products.

4.5.1 Flash Features for VIS Development

Flash supports the development of some typical graphic displays in VIS through its facilities (e.g., text, sound, video, animated graphics, etc.) and built-in methods (e.g., rotation, movement, etc.). Its scripting language, ActionScript (Donatis, 2006; Hamlin, Tarbell, & Williams, 2003) can be used to support interactive contents and enhance model presentation that make simulations to come *alive*. Table 4.5 relates VIS Graphic Displays to relevant Flash features.

Table 4.6 VIS Graphic Displays and Flash Features

VIS Graphic Display	Flash Feature
Abstract displays (e.g., graphs, histograms, etc.)	Flash runtime drawing methods such as *lineTo( )*, *lineStyle( )*, *beginFill( )*, *endFill( )*, *beginGradientFill( )*, etc. These methods can be written in an ActionScript class and associated with a movie clip symbol as a component.
Scale models	• Flash Drawing Tools • Flash import facilities to import various kinds of image and geometry files. Supported files include AutoCAD DXF (*.dxf), Silicon Graphic Image (*.sgi), JPEG Image (*.jpg), etc.
Static elements (e.g., servers or animated symbols)	A movie clip associated with an ActionScript file. The file controls *Keyframes* to animate the status of static elements.
Concrete dynamic displays	A movie clip associated with an ActionScript file. The movement of the movie clip onstage is controlled by a movie clip's instance's *_x* and *_y* properties.
Abstract dynamic displays	An animated movie clip that uses multiple frames and layers associated with an ActionScript file.
Tools for enhancing model presentation (e.g., audio, video and text)	Audio, video and other Flash Tools (e.g., *Text*, *Rectangle*, *Line*, etc.) and Flash built-in components (e.g., *Button*, *MediaController*, *Label*, *TextInput*, etc.).

4.5.2 Flash Component Construction

Flash supports architectures for component development. A Flash component is a compiled movie clip that contains a symbol that depicts its functionality and an ActionScript file that defines its operations as in Figure 4.5. Dragging and dropping this symbol onto the Flash stage will automatically create an instance of its class.

A component is often broken up to smaller components to reduce its implementation complexity. These smaller components are then tied to other components (e.g., through a *composition* technique) to form a more complex structure. By doing this, a component can now delegate relevant tasks to other components to perform the whole application functionality and this simplifies application development. In order to *encapsulate* its internal information and structures (i.e., its properties and behaviour), property accessing and behaviour triggering are only possible through *messages* specified by *signatures*; i.e., publicly accessible methods. This ensures that the component's internal modifications can extensively be made as long as its signatures are not altered.

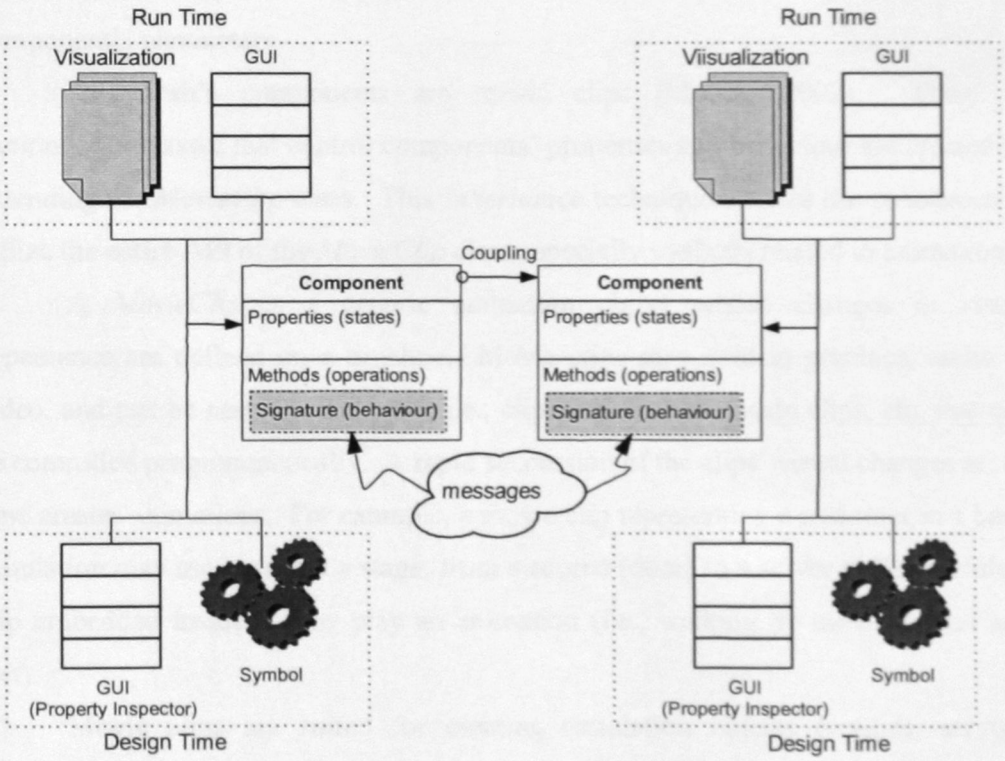


Figure 4.5 Component Architecture

Accessing component properties is typically through API. As an alternative, GUIs that compose of other types of objects (e.g., text boxes, combo boxes, buttons, sliders, etc.) are used as attractive platforms to parameterise the properties. In Flash, the interactive property changing can be done through *Property Inspector*. However, this facility is only available during design time and does not integrate any mechanism to verify input values (e.g., to force correct data types or limit the range of data values to avoid any logical errors). To address this, Flash allows designers to construct their own GUIs using the API approach either for filtering input data (e.g., displaying a warning for invalid data), easing data input processes (e.g., displaying step by step GUIs) or supporting component parameter manipulation during run time. Each GUI should be located in a relevant layer so that users can freely turn it on or off anytime they wish.

Showing the instant effect of data manipulations (e.g., scale, colour, description, etc.) on a component at design time can be done through a *Live Preview* facility. This facility can be utilized for providing interactive DES components that their current visual appearance can instantly be observed. Developers however need to embed the component with an external relevant movie file that consumes the component's parameters.

All Flash's components are movie clips (Moock, 2002). Thus, all ActionScript classes that control components' properties and behaviour are created by extending the *MovieClip* class. This *inheritance* technique enables the subclasses to utilize the entire API of the *MovieClip* class especially methods related to animations.

A *MovieClip* is a generic animation object whose changes in visual appearance are defined on a timeline. Movie clips may contain graphics, audio or video, and can be nested recursively; i.e., clips inside clips, inside clips, etc. that can be controlled programmatically. A rapid succession of the clips' visual changes at run time creates animations. For example, a movie clip representing a customer in a bank simulation may move across a stage, from a source (door) to a server (teller), while a clip embedded inside it may play an animation (i.e., walking by moving arms and feet).

Movie clips are suited for creating simulation objects (entities, servers, components, etc.) in DES. In addition to adding specific features, their classes should extend the *MovieClip* class to inherit its (1) properties (e.g., location, visibility, etc.), (2) methods (e.g., moving, rotation, etc.), and (3) built-in events (e.g., *click*, *rollover*,

drag and *drop*, etc.) to support interactive dialogues between users and the objects during run time; e.g., to change their parameters, to drag and drop the objects or their GUIs to other locations, etc. Furthermore, Flash allows tool developers to attach the objects with symbols to portray their functionalities. The use of appropriate symbols can help model builders to differentiate various objects and their tasks in a library.

Flash only considers components as reusable movie clips that simplify the creation of a Flash movie. Thus, many of its stand alone components (e.g., *Label*, *TextArea*, *DataGrid*, etc.) that do not offer cooperation with each other can be seen in its *Component Panel*. Such components do not suit the real definition of *component-based development* (CBD) that views components as customizable building blocks; each of which needs to offer specific services and can be aggregated visually or programmatically with each other to form an application. The aggregation could be through a coupling mechanism that wires components together using *interfaces*; i.e., ports that allow communications among them to perform the application's logic (see Figure 4.5).

The component approach suits DES model constructions since entities flow from component to component to receive different services. Analogue to these entity flows, signals can be used as activation mechanisms for certain components to support more complex DES; e.g., a transportation system. In this case, signals are sent by relevant components to activate transporter or conveyer components.

We focused on the development of DES components and approaches for wiring them together and manipulating their parameters during runtime. Combining these approaches and the facilities that allow learners to view component states using various data visualization tools may offer advantages especially in easing learning. Details about this are discussed in Chapter 5.

4.5.3 Other Advantages of Flash and Its Drawbacks

Besides supporting architectures for component development, Flash offers other advantages for building VIS models particularly and any types of simulations generally over other multimedia-development applications. These include:

- Flash makes it easy to animate smooth motion of simulation entities at a default rate of 12 frames per second (fps). This is adequate for web-based animations, but model users can easily change this to control the animation speed (i.e., up to 120 fps). Higher rates smooth visual changes but will increasingly tax the host CPU. Slower rates reveal more detail, but may make animations less smooth. Note that this specified fps value only acts as the maximum speed limit; i.e., the animation should not play faster than the fps value. However, the minimum limit of its execution is uncontrollable since it depends on CPU speed.
- Flash animates a sequence of images using key frames. Each key frame can represent a critical point of animation; e.g., the change of shapes or visual appearances.
- Flash offers a large stage for drawing and composing objects and playing animations. Its run-time player offers the ability to pan, zoom out and zoom in to look at interesting locations around the stage.
- Flash employs *vector graphics* that use line segments to form figures. Thus, these figures can be scaled without loss in resolution and clarity. However, *raster graphics* that represent images as an array of pixels are still supported.
- Flash produces executable files that can be played on both PCs and Mac platforms. These files can be distributed via Internet without any modifications.
- Flash allows model builders to control the *visual depth* of an object. This eases the arrangement of various simulation objects and their GUIs on a stage.
- Flash provides some supports for student assessments (Castillo et al., 2004). Teachers can use these to create exercises that gauge students' understanding of a certain topic.
- ActionScript syntax is similar to Java; which again similar to the C family. For those who are familiar with these languages, ActionScript can be learnt without much effort.

Besides these advantages, Flash also has some drawbacks; i.e.:

- Flash is not supported on Apple mobile devices. This limits the delivery of Flash-based contents to Apple tablets and the Iphone. However, there are now some

applications (e.g., *iSwifter*) which claimed to run Flash contents directly on the Ipad and Iphone.

- Flash applications require an updated plug-in to play. Downloading the plug-in may consume time.
- Flash applications may be slow to download. This situation may frustrate users with slow bandwidth or internet speed.
- Flash applications cannot be indexed by most search engines. This may limit its visibility or rank in web browsers.
- Flash applications should be developed to serve a specific purpose of its site. The use of Flash to only decorate a webpage will annoy users and cause them to leave the site.

4.6 Flash Components for Queuing Systems

Based on the process-oriented modelling style (Castagna, 1997; Craig, 2007; Garrido, 1999, 2001), we have structured an overall class diagram for creating Flash-based components that can be used to construct animated queuing models as in Figure 4.6. This structure is the combination of the class diagrams discussed in Chapter 3 with some additional classes.

We extend all these classes from the *MovieClip* class for two reasons. First, extending the *MovieClip* class allows us to utilize its *built-in events* to provide drag and drop and interaction environments during runtime. By default, Flash allows its components to be dragged and dropped at authoring time. However, supporting this capability during runtime needs us to implement the *startDrag(this)* and *stopDrag(this)* events in relevant classes. Allowing learners to have their own model GUIs through creating, customizing and positioning visualization components is important for learning (Ebner & Taraghi, 2010). The same thing applies to providing an interaction environment where the *onRelease( )* event is used for accessing component GUIs during runtime. Second, extending the *MovieClip* class allows us to rightly control the depth of each component instance on the stage using the *createEmptyMovieClip(instanceName, depth)* method. For example, entity instances should have smaller depth values compared to other movie clips to guarantee that they

are always on bottom of other component instances; e.g., visualization components. This method can also be used to create a container; on which other movie clips (e.g., textboxes, buttons, labels, etc.) can reside. This ease the construction of component GUIs since the depth of their child is now controlled by its parent and dragging the parent movie clip to other locations will automatically retract its entire child.

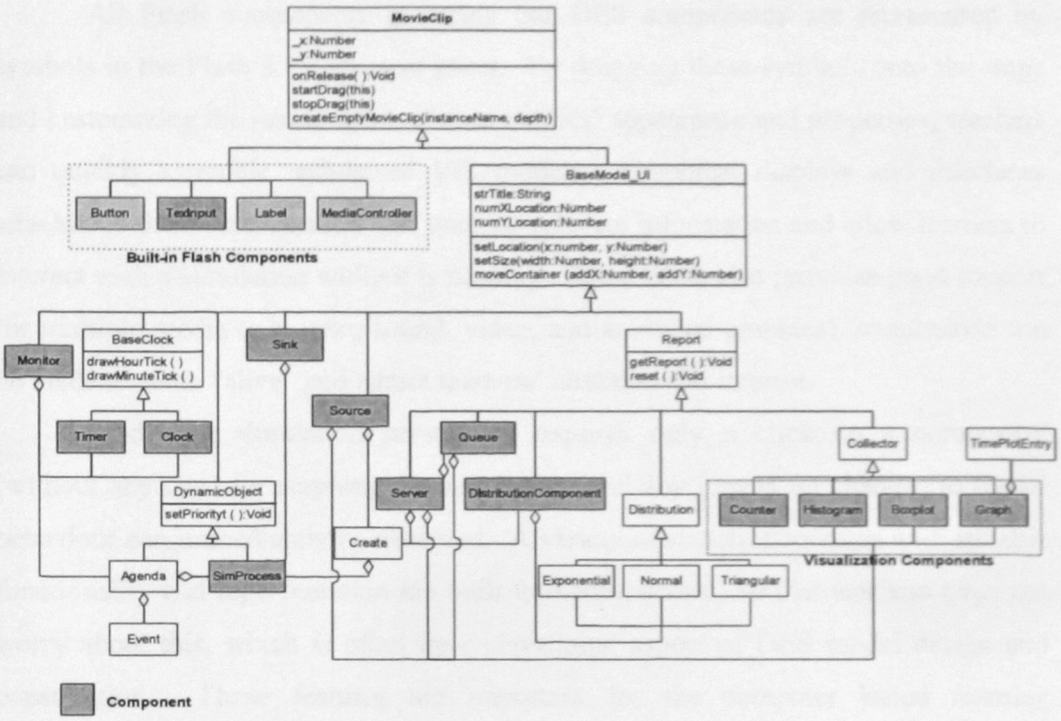


Figure 4.6 Class Diagram of *Components* for Simulation *Input and Output*

We designed and created three other components; i.e., the *DistributionComponent*, the *Source* and the *Sink* components to ease DES model constructions. The *DistributionComponent* is used to provide a combo box of a list of distribution types. Its main purpose is to ease the selection process of random samples in other components; e.g. the *Source*, the *Queue*, the *Station* and the *Server*. The *Source* component is a component that receives parameters that control the creation of entities; e.g., time for the first arrival, time between arrival, priority, entity type, etc. These parameters are fed to the *Create* class through a composition technique. In order to generate entities appropriately, the *Create* class has to compose two classes; i.e., the *SimProcess* class to create entity instances and the

DistributionComponent to control the creation of entities based on a specified distribution type. Since code for creating entities has been embedded in the *Source* component, model builders do not need to write any code to perform this task as in any simulation languages. The *Sink* component is to destroy the *SimProcess* instances that have been created so that computer memory allocated for these instances can be freed and reclaimed by the Flash's garbage collector.

All Flash components including our DES components are represented by symbols in the Flash's *Component* panel. By dragging these symbols onto the stage and customizing the resulting simulation entities' appearance and properties, teachers can quickly assemble web-based VIS models. Graphical displays and interfaces attached to these entities show and animate relevant information and allow learners to interact with a simulation while it is running. Since Flash also provides good support for multiple media (e.g., text, sound, video, and animated graphics), simulations can be made to come "alive" and attract learners' attention and interest.

Modifying simulation parameters requires only a click on a component (without any need for stopping the simulation) and any impact on changes to model behaviour can immediately be observed. A variety of statistics counters with suitable functionality and representation are built into components, so that teachers need not worry about this, which is often time consuming aspect of DES model design and construction. These features are important for the computer based learning environment (Min, 2003). Since the components have been developed in Flash, VIS models can draw on its functionality to easily integrate with a learning management system (LMS). Access from remote locations through internet browsers is a further benefit that can be attributed to this architecture.

Table 4.7 shows three types of Flash DES components that have been developed for supporting the construction of queuing networks. All three types of queuing networks are supported: *open queuing* that studies a system in which transactions are generated, flow through a model and disappear (e.g., in most service systems), *closed queuing* that examines a system in which transactions are permanent (e.g., in a computer system) and a mixture of open and closed queuing (e.g., in a healthcare system). The functionality and features of these components that support both teachers and learners are detailed in Table 4.8. Figure 4.7 meanwhile shows the location of DES components in the Flash *component panel*.

Table 4.7 DES Component Types

Component Type	Description	Example
Active	Components which involve cooperation with entities	Source, Queue, Sink, Monitor, Server
Passive	Components which do not involve cooperation with entities	Station
Visualization	Components which show states of active components	Counter, Graph, Histogram, Boxplot

Table 4.8 Flash Components for Building a DES Model and Their Functionalities

Component	Functionality/Feature
Source	Animates the arrival of entities. • Teachers can specify the time of the first entity's creation, priority value and the default distribution of time between successive arrivals. • Learners can click on the Source's instances, pick a list of available distributions and change the default parameter of entities' time between arrivals. They can directly observe the effect of the changes to the model's behaviour. Each instance automatically collects and displays the number of entities that have entered the model at the current simulation time.
Queue	Graphically animates queues with priority rules such as FIFO (First In First Out), LIFO (Last In First Out), lowest priority value, highest priority value, or a random order. The removal of entities from a queue is controlled by the priority rule at the time of removal. All Queue instances automatically collect statistics, such as the number of entities which have left a queue, maximum, minimum, sum, mean, variance and standard deviation of times spent in the queue. • Teachers can initialize a default priority rule and specify what visualization instances will report queue statistics. • Learners can change a queue rule anytime time they wish and observe the effect of priority rules on a model's behaviour through the changes in queuing statistics.
Sink	Collects and graphically displays entities leaving a model. • Teachers can attach visualization instances to display statistics about time entities spent in a model. • Learners can mouse over a Sink instance to obtain maximum, minimum, sum, mean, variance and standard deviation statistics for times entities spent in a model.

Station	Represents points to which entities are transferred in a model; i.e., points on the stage they can move to.
Distribution	Generates random samples from a list of specified distributions. • Teachers can use this component to sample the duration of various model-time consuming activities.
Monitor	Acts as a simulation engine and controls viewing ratio and simulation length. • Teachers can initialize viewing ratio and simulation length. They can also link Clock and Timer instances to graphically represent simulation current simulation time and its proportion to simulation length respectively. • Learners can click the Monitor's instance to observe simulation events that have been executed, a current event being executed, and the list of events still to be executed in future. They can also stop and resume animations and adjust animation speed by only clicking appropriate sub-symbols.

Figure 4.7 Flash Component Panel

Simulating DES entities in the Flash environment requires model builders to create an ActionScript class that extends our *SimProcess* class. The class describes the entities' lifecycles using *if-else* or *switch-case* statements. This task could not be avoided since ActionScript does not support *coroutines* or *threads*. Adobe's official reasons for this are that threads will induce very different behaviour on different

machines especially in performance intense platforms and race conditions in threading will led to performance problems on the Flash player.

Flash imposes model builders to convert an image to a movie clip symbol before it can be attached to an ActionScript class in order to animate these entities. For example, to associate a movie clip with a *Customer* class that extends our *SimProcess* class, the following actions must be stepped through:

1. Draw a picture on (or import a picture in any format onto) the Flash stage.
2. Convert the picture to a symbol and give it a name. This symbol will appear in the *Flash Library*.
3. Select a movie clip symbol in the *Flash Library*.
4. Right-click on the symbol and choose "Linkage".
5. In the resulting dialog, enter the symbol's name (e.g., *Customer*) and its associated class (i.e., the *Customer* class).
6. Select "Export for ActionScript" as "linkage type".

Once the movie clip is in the Flash library, we can make the *Customer* objects' visual appearance more attractive by providing keyframes named *onMoving*, *inQueue* and *inProcess* to depict the *Customer*'s states. All code that animates these states together with code to handle their movement from component to component and halt at a queue or being processed by a server has been defined in the *SimProcess* class. Note that these frames are defined on the *Customer* symbol's timeline and *not* globally on the stage. This gives us a local animation for *Customers* (i.e., their change of appearance in different states) that is nested inside the main animation (tween movements across the stage). To create this local animation, we must step through the following actions:

1. Right-click the *Customer* movie clip symbol in the Flash Library and select *Edit* from the resulting pop up menu.
2. Select frame 10 on the timeline.
3. Select *Insert* > *Timeline* > *Keyframe*.
4. In the *Properties* panel, change Frame *Label* to *onMoving*.
5. Draw a suitable picture of a customer's movement on the current Flash stage.

6. Repeat steps 3 to 5 for frame 20, 30 and 40, and make appropriate changes at each step.

The entity movie objects can be clicked during execution time to display a variety of relevant information; e.g., its number in a model, its creation time, the time spent in queues or servers that it has visited.

The server objects can be animated in a similar way; i.e., by assigning different symbols to keyframes *Idle* and *Busy* and attaching each symbol to our *Server* class. Note that we leave this task in the hands of model builders instead of providing a compiled *Server* clip in order to give them flexibility in animating server objects using any images they wish. Actually, a set of *Server* components with different symbols can be provided. The server's capacity and service time can be changed during a simulation run by clicking its symbol and then picking up one type of distribution from a list of available distributions.

4.7 Flash Components for Visualizing Queuing Systems

Table 4.9 shows Flash components for visualizing model states and their functionality. Figure 4.8 meanwhile shows some sample instances of visualization components (e.g., *histogram*, *graph*, *boxplot* and *timer*) on the Flash stage during a simulation run. Visualization components are connected to active components (i.e., *Source*, *Queue*, *Sink* and *Server*) through a composition technique (see Figure 4.6).

Embedding visualization components in an active component through a hard-coded composition approach has two distinct drawbacks. First, this approach requires us to explicitly declare the name of the visualization instance in the active component's class variables so that we can access its methods and properties and update its states. This problem is getting worse if we want to embed many types of visualization instances to provide a platform for learners to flexibly create various visualization tools during runtime.

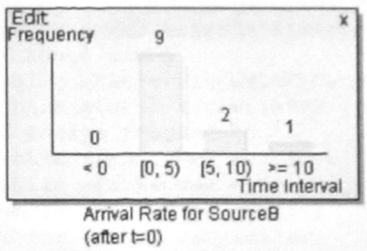
We can use an array to store each type of the visualization instances. However, an array is not a suitable data type for storing such a variable size of visualization instances since in certain languages this may cause space wasting (if we

do not fully use the array's size) and an insertion problem (if the array size has been exceeded). We can alternatively store a list of array objects of type *Collector* (see Figure 4.6) or general objects, but treating a base-class object as a derived-class object is a bad programming practice and may cause errors; e.g., when we cast a base-class as a derived-class and then refer to derived-class members that do not exist in that object. Second, this approach tends to contribute to syntax errors since any modification of the visualizations' method or property names will impose the changes of code in the active component's class.

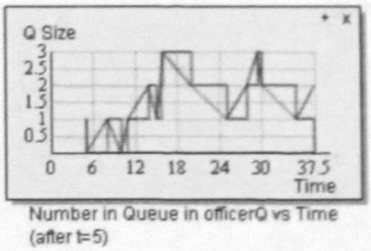
Table 4.9 Flash Components for Visualizing DES Models and Their Functionalities

Component	Functionality/Feature
Graph	Dynamically animates patterns of changes in simulation outputs, such as the current number of entities in a queue versus simulation time, or the number of a server's *busy* units versus simulation time. • Teachers can specify width and height, a title, a colour for graph lines, background and fill area for each Graph's instance. • Learners can clear the previous data, drag the Graph's instances to any location and resize them at any time they wish.
Histogram	Dynamically animates frequency information, such as the time spent by entities in a queue, the operation time of a server, the time between arrivals, the successive time between departures, etc. • Teachers can specify width and height, a title, a colour for text, background, bar fill area, maximum value, minimum value and the number of intervals. They can also activate drop-shadows for each instance of the Histogram component. • Learners can change maximum value, minimum value and the number of intervals at any time to see a new distribution of frequency information, drag the Histogram's instances to any location and resize them at any time they wish.
Boxplot	Dynamically animates groups of numerical data through its five-number summaries. It is a complementary tool for the Histogram component. • Teachers can specify width and height, a title, a colour for graph lines, background and fill area for each Boxplot's instance. • Learners can drag the Boxplot's instances to any location and resize them at any time they wish.

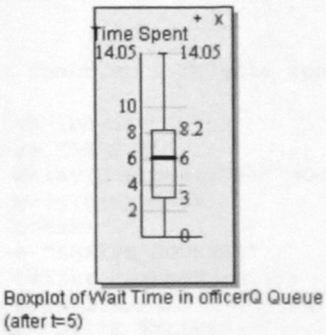
Clock	Dynamically animates the current simulation time while a simulation is running. • Teachers can specify a title, fill colour, initial time value and simulation time unit. • Learners can drag the Clock's instances to any location and resize them at any time they wish.
Timer	Animates the proportion of the current simulation time to its total duration. • Teachers can specify title, fill colour and elapsed time fill colour. • Learners can drag the Timer's instances to any location and resize them at any time they wish.



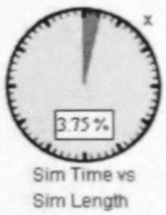
(a) Graph



(b) Histogram



(c) Boxplot



(d) Timer

Figure 4.8 Samples of DES Visualization Tools

4.8 Example

This section presents a simple example of how the DES libraries and components may be used to model a queuing scenario. The example simulates a bank, where customers *arrive*, *walk* to a counter, get *served* by a teller and finally *exit* from the bank. The corresponding model uses a single *Server* object for the teller, a stream of

SimProcess instances representing customers, and a number of active and visualization components for structuring the model and visualizing its states. As mentioned earlier, some active components embed *Distribution* objects for sampling the duration of various model-time consuming activities.

To represent customers, we must first create a new ActionScript class and save it under an appropriate name (in this case *Customer.as*) to the *simulation tools* folder. Here we define a *Customer* class based on the *SimProcess* class, declare various class variables and define its *lifecycle* method; see Listing 4.1.

```
1   // import packages
2   import Monitors.*;
3   import Resources.Server;
4
5   class Customer extends SimProcess {
6     // route times
7     public static var walkToCounterTime;
8     public static var walkToExitTime;
9     // active components
10    public static var myEntry;
11    public static var myBench;
12    public static var myExit;
13    public static var teller;
14
15    private function init ():Void {
16      addPhase("ARRIVAL, ARRIVE_COUNTER, SEIZE_TELLER, DELAY_TELLER,
17      RELEASE_TELLER, DISPOSE");
18    }
19
20    public function lifeCycle (phase) {
21
22      switch (phase) {
23        case "ARRIVAL":
24          delay(Customer.walkToCounterTime.sample());
25          moveTo(myBench);
26          break;
27       case "ARRIVE_COUNTER":
28          teller.request(this);
29          break;
30       case "SEIZE_TELLER":
31          delay(0);
32          moveTo(teller);
33          break;
34       case "DELAY_TELLER":
35          delay(teller.serviceTime.sample());
36          break;
37      case "RELEASE_TELLER":
38          teller.release();
39          delay(Customer.walkToExitTime.sample());
40          moveTo(myExit);
41          break;
42      case "DISPOSE":
43          myExit.remove(this);// remove this object
44          break;
45      } //end switch
46    }
47  } // end Customer class
```

Listing 4.1 The Customer Class

In lines 7 and 8, we declare two class variables for representing customers' route times; i.e., a *walkToCounterTime* distribution for sampling the time taken by customers to walk from an entry to a counter, and a *walkToExitTime* distribution for sampling walking time from the counter to exit. In line 13, we declare a *teller* variable representing an object of the *Server* class. Note that visualization components (e.g., Graph, Histogram, Boxplot, etc.) can be composed to the active component instances using the Flash's *Properties* panel. Line 10 to 13 stores instances of *Source*, *Queue* and *Sink* component respectively.

The *init* method (line 15) initializes *Customer* objects. Here we must specify a sequence of phases (i.e., a *lifecycle*) that all Customers instances step through. The *addPhase* method in line 16 attends to this requirement. The *lifecycle* method's description begins with a description of what will happen when the control returns to this object, based on the phase it is in (lines 23 to 44). *Customer* objects are generated by a *Source* instance based on specified time between arrivals. Upon *arrival*; i.e., the first phase of the lifecycle (line 23), a *Customer* object advances itself to the next phase by calling *delay*. The *Source* instance (i.e., *myEntry*) instantiates a new *Customer* object, whose associated movie clip is then used to animate it on the stage. *delay* (line 24) schedules the current customer to continue to its next phase and inserts a corresponding event notice at the appropriate point on the *agenda*. At the right model time instant, the monitor will later remove this event notice from the head of the *agenda*, retrieve the associated object and direct it to continue its execution from the relevant point on its lifecycle. The *monitor* will terminate the simulation when the end of the requested duration is reached or when no more events can be found on the *agenda*.

In preparation for the model's animated display, the location of the *Source* instance is the initial location for arriving *Customer* objects and the *moveTo* method (e.g., in line 25) moves a customer's picture to a given location (e.g., that of a *Server* object). While the previously described actions prescribe simulation activities, this method serves animation. Note that *moveTo* uses a motion tween, whose duration is controlled by the ratio of animation to simulation time, a value that can be dynamically adjusted by the model users.

Server objects have two methods: *request* and *release*. *request* (line 28) allocates any free unit to a requesting customer. If all available capacity has been used, a *Customer* object has to wait in a queue. A call on *release* (line 47) reactivates

a *Customer* object, returns however many capacity units it holds, and gives the next waiting customer a chance to acquire those units. In the final phase of a *Customer* object's lifecycle, the *remove* method (line 43) destroys the *Customer* object, whose storage will eventually be reclaimed by the Flash's garbage collector.

Notice that we had to use a switch case statement to execute different sections of code, based on the *phase* a currently executing instance of the *Customer* class was in. *Phase*'s value was stored in a *phase* attribute and the *addPhase* method listed six valid phases (i.e., *ARRIVAL, ARRIVE_COUNTER, SEIZE_TELLER, DELAY_TELLER, RELEASE_TELLER* and *DISPOSE*). While this construction is arguably a rather clumsy way to implement a process oriented modelling framework, it was forced by ActionScript 2's lack of support for either *coroutine*, *threads* or any other control abstraction which would allow the persistence of state that could store one of multiple entry points to a method.

In addition to *Customer* objects, which arrive, request services and leave, we need to specify the environment these dynamic objects are to operate in; i.e., we need to add relevant components to the Flash' stage (see Figure 4.8), specify their names and link the visualization components to the active components. We then initialize the active components' properties; e.g., simulation length, server capacity, time-between arrival, etc.

To complete our model's definition and use the *Customer* class, we must first create a new Flash document. For this example, we need just two keyframes: *Parameter* and *Animation*. The *Parameter* keyframe displays a form for choosing statistical distributions for *Customer* objects' route times. *Distribution* components are dragged from the *Components* panel and dropped at appropriate places on the Flash's stage. They are then used to initialize the Customer's *walkToCounterTime* and *walkToExitTime* variables. This keyframe can be ignored if model developers choose not to give model users flexibility in customizing their own *Customer* objects' route times.

The *Animation* keyframe is used as a stage to assemble the visual representation of the model's animation. Here we use active components (i.e., *Source*, *Server*, *Queue* and *Sink*) and visualization components (i.e., *Timer*, *Clock*, *Graph* and *Histogram*), whose properties (e.g., time between arrival, service capacity, colour, width, etc.) can be changed through a *Component Inspector*. For each change in properties, the component's appearance on the stage will be automatically adjusted.

Note that each component should be given a unique identifier that corresponds to the names used by the *Customer* class (e.g., *myEntry*, *myBench*, etc.; see line 10 to 13) to make sure that these variables are correctly assigned with their relevant component instances. To animate the *Customer* and *Server* objects, the approach discussed in Section 4.6 needs to be followed. A model layout as a base for model structures and animation can either be drawn using Flash's drawing tools, or we can import external graphic files in JPEG or DXF formats.

Figure 4.9 shows an example of a VIS model built using our DES components. It is indeed the model constructed using the previous code and procedures, with an addition of one more *Source* and *Server* instances and another class of entities. These entities need two servers, the second of which is the same one that processes the *Customer* objects. As shown in this figure, learners can change the distribution of time between arrivals, server capacities and service time and queue priority rules (queuing disciplines) by clicking relevant component instances. Figure 4.10 meanwhile shows sample information that can be obtained from the underlying VIS model. This includes statistics on queues and servers, as well as what previous events have been executed, what current event is being executed and what further events are still scheduled for execution.

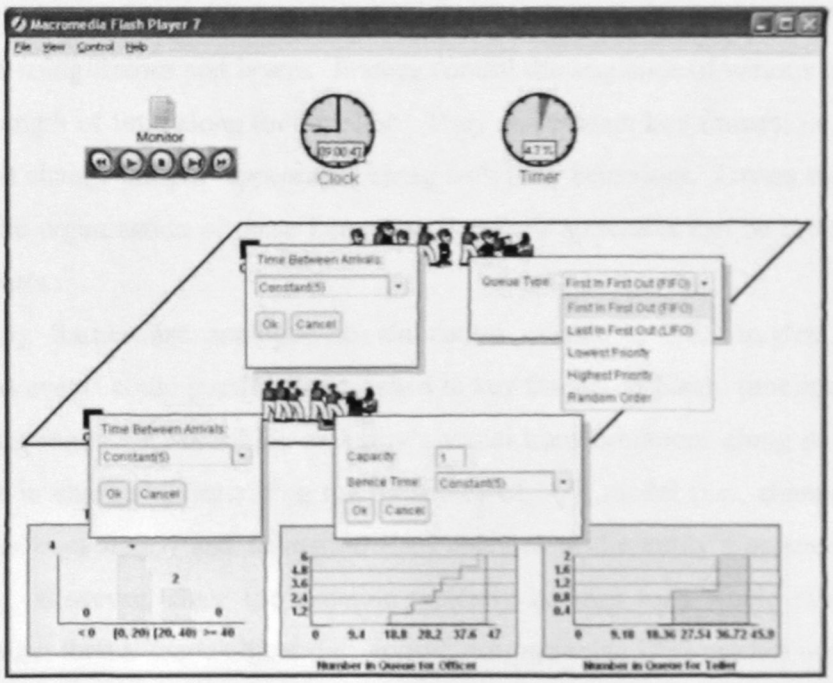


Figure 4.9 Sample of Interactions between Learners and a Model

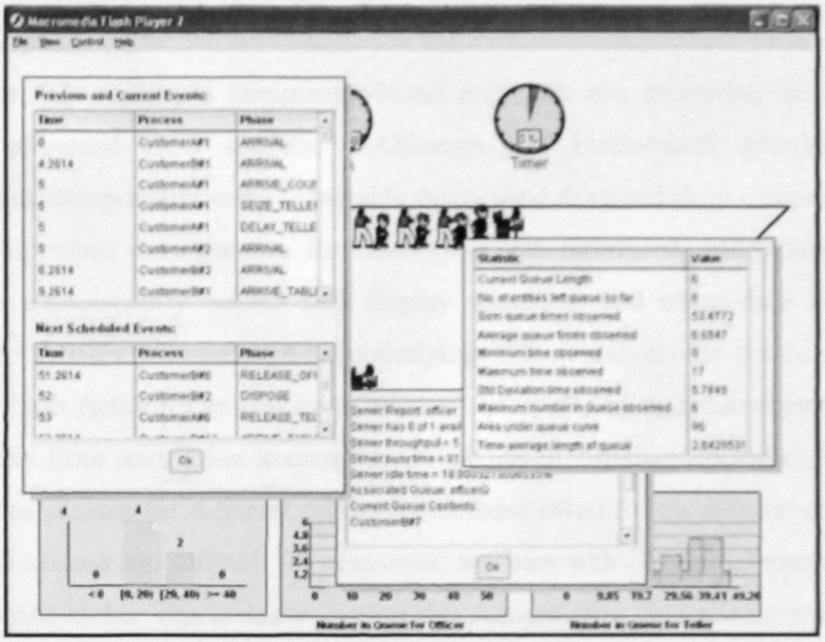


Figure 4.10 Sample of Information Gained from a Model

4.9 Problems and Pitfalls

Flash controls movie contents over time using a timeline. Rapidly running the timeline forms an illusion of animated images. All animated images in Flash are organized using frames and layers. Frames control the sequence of various images in definite length of time along the timeline. They can contain key frames; i.e., control points that change images' appearance along with their behaviour. Layers meanwhile support the organization of these images so that their structures can be broken up to smaller parts.

Key frames are analogue to simulation events in DES models. Thus, simulation events could possibly be attached to key frames on Flash' timeline. In this fashion, an animation describing an entity's visual transformations along its timeline would be in charge of describing the dynamics of both model (i.e., changes in the entity's abstract states) and animation (i.e., changes in the entity's appearance and location). However, since the timeline typically belongs to a whole movie (i.e., model) rather than a single object (i.e., entity), programming DES models on this way is unpractical. The use of timeline to stage model and animation methods (e.g., its

movement, rotation, etc.) to control the object's behaviour will make model code unmanageable.

For this reason, a component-based approach can minimize the effort of creating animated DES models. Although our Flash-based simulation and visualization components strive to provide easily used drag-and-drop components and visually supported environments for developing VIS interfaces, and although these interfaces automatically collect and display statistical and other data and allow learners to flexibly interact with an underlying VIS model, model builders need to program Flash ActionScript classes to annotate lifecycles of dynamic objects (i.e., to flow entities from component to component) and attach visualization tools. In spite of the fact that classes for different types of processes often follow a common pattern, this is still tedious and difficult for occasional teachers with little programming skills. As mentioned earlier, this is due to the fact that ActionScript does not support suitable semantic abstractions for providing a *coroutine* feature. While we believe that our first iteration of a Flash-based "DES with animation" toolbox is a step in the right direction, its use is still short of the level of ease that we hope to achieve.

Ideally there should be no need for Actionscript coding at all, so that models and animations could both be constructed by dropping and linking components from libraries while cloaking them in appropriate visual representations. Unfortunately Actionscript currently offers no support for turning text into code (i.e., there is no equivalent to an *eval* statement) and a small compiler would need to be written to allow users the flexibility to alter dynamic components' behaviour through visual interfaces. In Chapter 5, we introduce one approach for building interactive visual components that will cater the current need to annotate the lifecycles of dynamic objects and easily connect the components.

The main tricky issue in integrating an animated simulator to a DES model is to correctly trigger sorted events based on a viewing ratio specified by learners (i.e., to stop, continue or proportionally decrease or increase model time before attempting to trigger next events in the *Event List*) since they are free to stop or change the ratio at any time they wish. This includes precisely animating two consecutive events at appropriate time and moving entities within specified time frames. In the Flash environment, animating such entities' active and passive states can be accomplished using the *setInterval* and *clearInterval* functions.

We use the Flash's *setInterval* function that periodically calls a move method to update an entity's locations during its movement to a target location (see Equation 4.1 and Equation 4.2 in Section 4.4) and the *clearInterval* function to clear the interval once the entity has reached its destination. Flash claims that this function is accurate since it is not influenced by any frame rate values and can thus be used to update object properties at a specified time interval. To check this, we conducted some tests and found that it was only 2% to 6% different for one second interval in various frame rates. Tests on other machines also confirmed the claims in spite of the fact that the execution of frame rates depends on CPU speeds.

However, a pitfall occasionally arises when a viewing ratio for a certain model (changed using a *slider*) reaches at a certain value. This is especially true when we want to update an object's locations in very small steps (that typically needs a very small interval time; i.e., less than a second) so that it can move smoothly. For example, let say the distance between two locations is 10 distance units and its route time is 2 time units. If we assume that a viewing ratio is 1, the entity then needs to reach its destination in 2 seconds. Since it only needs 2 movement steps (i.e., 5 distance units for each second), the animation looks jumpy. To make it look smoother, we need a smaller time interval so that we can get smaller steps, but still within 2 seconds time frame. For example, if we use a 100 milliseconds time interval, we can have 20 steps with each step causes 0.5 increment from its previous location. If users increase a viewing ratio, the time interval must be decreased; e.g., for a viewing ratio value of 2, the interval should be 50 milliseconds since model time must be maintained but animation is now changed (so that the object can reach the target location in a second animation time, refer to Equation 4.1). However, we notice that entities do not exactly reach at their target locations within specified animation time, making our animation engine looks like it is not working accurately.

We found that the *setInterval* function will only start executing a called method after it has completely finished executing the previous called method. This problem becomes worse when a called method has intensive code that needs an amount of time to be processed (e.g., it contains repetition structures) or when the animation is running in slow CPUs. As a result, the elapsed time of the handler function gets added to the overall interval, making accumulated delays in executing the method within the specified time frame. In our case, this delays the update of objects' locations and consequently delays the arrival of the object. To cater this

problem, we checked the time elapse and adjusted the motion accordingly based on that current time.

4.10 Extensibility

Figure 4.11 extends the overall class diagram in Figure 4.6 to support DES for logistic and manufacturing systems. As discussed in Section 3.5, logistic systems require two types of objects, i.e., *Bin* and *Stock* while manufacturing systems require two types of objects, i.e., *Transporter* and *Conveyer*.

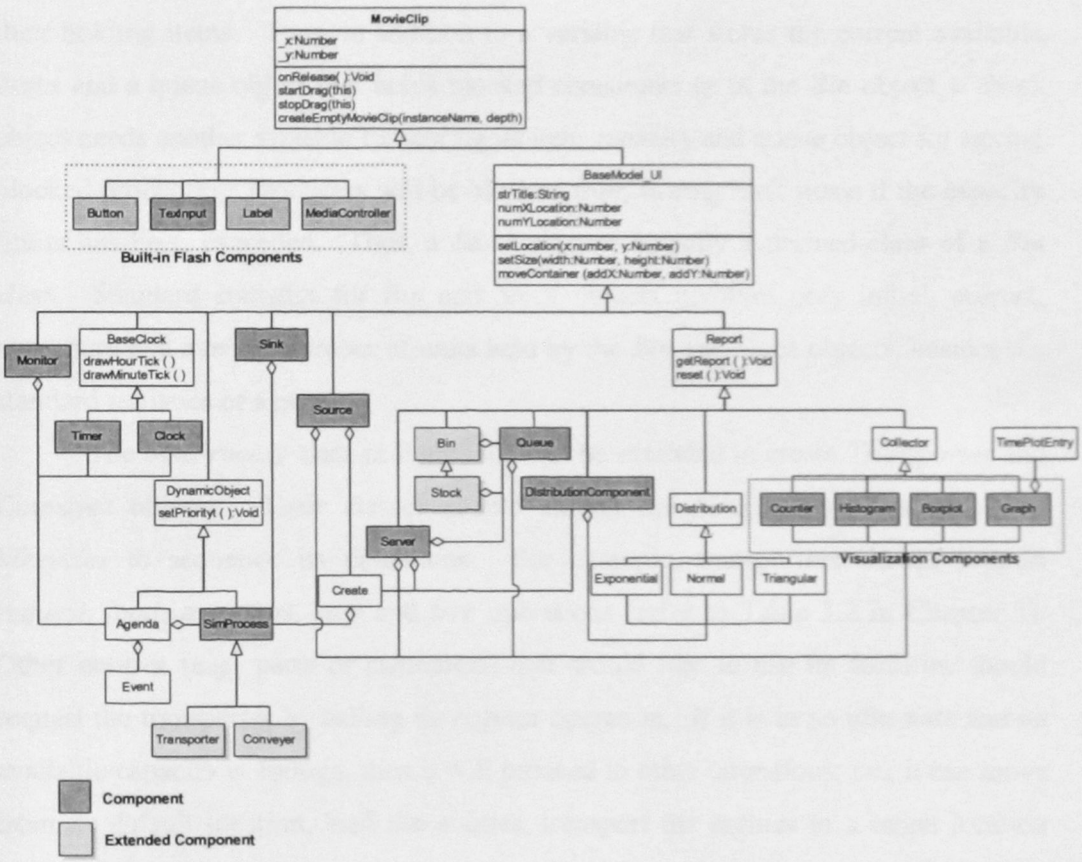


Figure 4.11 Extended Components for Supporting Logistic and Manufacturing Systems

If we compare the patterns of synchronization in producer/consumer relationships with capacity constrained resources, their operations are similar. A

Server object stores a number of units to be allocated for competing processes (i.e., the *SimProcess* objects) and takes back the unit(s) once they have been released. Thus, we need to declare a variable in a *Server* class for storing the current available units and a *Queue* object for holding requested processes.

In a *Bin* object's operations, a producer deposits items through a *store* operation while a bin object supplies the stored items for requested consumers through a *deliver* operation. If the stored items are not enough, consumers must be queued and will be treated using a FIFO rule. Thus, a *Bin* class also needs to declare a variable for storing the current available items and a *Queue* object for holding blocked consumers.

Compared to *Bin* objects that can store unlimited items, *Stock* objects limit their holding items. Thus, in addition to a variable that stores the current available items and a queue object that holds blocked consumers as in the *Bin* object, a *Stock* object needs another variable for storing its item capacity and queue object for storing blocked producers. Producers will be blocked from storing their items if the capacity limits has been exceeded. Thus, a *Stock* class is actually a derived-class of a *Bin* class. Standard statistics for *Bin* and *Stock* objects involves only initial, current, maximum and average number of units held by the *Bin* and *Stock* objects, besides the standard statistics of a queue.

The *SimProcess* class in Figure 4.5 can be extended to create *Transporter* and *Conveyer* objects. Their classes should extend the *SimProcess* class and have *lifecycles* to sequence its operations. For example, transporters should support *request*, *load*, *transport*, *stop* and *free* operations (refer to Table 3.2 in Chapter 3). Other entities (e.g., parts or customers) that would like to use its facilities should request the transporter by calling its *request* operation. If it is in an idle state and its available capacity is enough, then it will proceed to other operations; i.e., it can move from its default location, load the entities, transport the entities to a target location based on its velocity, and stop and release the entities when arriving at its destination. Otherwise, the requested entities need to be hold in a queue until both conditions are true. For this, we need a queue that listens to the transporter' states; e.g., by receiving the transporter's signal message. Ability to send and receive signals to or from other types of objects (that notifies a certain event has happened in the object) is a better communication approach among objects that enables us to provide a component that handles these processes internally.

However, using a composition technique to achieve such a communication between classes (i.e., by storing other instances) without implementing a relevant mechanism is not a suitable approach. For example, a tool designer needs to ask one type of objects to regularly check if its interested objects change states; and this process will incur execution penalty. As there are many other types of objects that are interested to listen to a single source object, the programming process is getting harder since the synchronization process is getting complex. In Chapter 5, we will introduce such an interaction between DES components that allows us to flexibly registered interested objects to an object, while maintaining a loose coupling between these components.

CHAPTER 5

COMPONENT-BASED MODELING FOR ANIMATED SIMULATION

5.1 Introduction

Ease of use and flexibility are essential criteria for DES tools. Unfortunately, both often conflict with each other. General-purpose DES simulators such as *PSim-J* (Garrido, 2001), *SSJ* (L'Ecuyer et al., 2002), *J-Sim* (Kacer, 2002), *DESMO-J* (Meyer et al., 2005b) and others can be difficult to master, since they typically require significant programming effort for model construction. Visual and interactive commercial modelling tools; e.g., *Arena* (Kelton et al., 2004) and *ProModel* (Harrel & Price, 2003) offer a user-friendly environment for construction and initialization of simulation models. Unfortunately, they often lack flexibility since their architectures are hidden and difficult to extend with additional simulation logic.

Although object oriented simulation libraries have long been used in providing a flexible and powerful simulation environment, they do not usually promote ease of use. Component-based simulation tools that provide links between simulation libraries have been proposed to solve this problem and have been adopted by commercial simulation tools and other complex software.

Our primary focus is to design and construct easy-to-use and extensible DES simulation tools that foster *learning through insight*; i.e., models that improve understanding through observation. Such models should incorporate *interfaces* to visualize model structures, *activities* to request interactions and challenge learners' understanding, *interesting scenarios* to attract learners' activities and challenge their imagination, *animation* to depict processes and dynamic behaviour, informative and meaningful *feedback* to reflect learners' actions and motivate them for further experimentations and *saving ability* to record interesting scenarios. The runtime interaction demands the implementation of *concurrent animations* to immediately

display the effect of changes rather than *post-processed animations* or *direct simulation-animation* (Hill, 1996).

Based on the benefits offered by component technologies and the importance of animations and visualizations in learning, we have identified two design patterns (i.e., generic solutions to systematically structure classes in object oriented applications) that are useful for the construction of interactive DES components. These patterns are the *Delegation Event Model*, which is used to link components together, and the *Model-View-Controller (MVC)* pattern, which is used to support GUIs and multiple visualizations of component states for providing a complete picture of model performance over time.

In Chapter 4, we designed and constructed DES components using *Flash ActionScript* (Moock, 2004). Besides its strengths as an animation tool (Mohler, 2006) and its support for component design (e.g., a default GUI, live preview, symbolizing a class, packaging facilities, etc.) and cross-platform distribution (i.e., through the WWW) and integration (i.e., through LMSs), a sample of ActionScript basic classes and interfaces (i.e., a group of related methods with empty bodies that defines common functionalities across various classes) for implementing many useful design patterns are also well documented (e.g., see Lott & Patterson, 2007; Sanders & Cumaranatunge, 2007).

This chapter presents the concepts related to the design and development of our interactive DES components for eliminating the need to write entities' lifecycles during design time and supporting the creation of various model visualizations during runtime. We first review the principles of component-based simulation. We then relate these principles with our model architecture to provide a graphical environment for building, visualizing and experimenting with the models. The strengths and weaknesses of some existing component-based simulators are also discussed. Based on the architecture, we identified the combination of two design patterns that fit the design of interactive DES components; i.e., the *Delegation Event Model* used to forge links between DES active and passive components and the *MVC (Model-View-Controller)* pattern used to loosely couple between components, their GUIs and their visualizations to provide facilities for model customization. The explanation of how both design patterns can be implemented in the Flash environment (including interfaces and classes that are used to create our components and their connections) is also presented. This chapter continues with the discussion of how to store a model's

states so that its visualizations and parameter settings can be saved for future use. To show the benefits of the combination of both the design patterns in providing a truly attractive and interactive environment, an example of a DES model is then presented. We further our discussion on how to cater with the model complexity by partitioning the model. This chapter ends with some discussions of problems and challenges that we faced during the design and implementation of our DES components.

5.2 Component Based Simulation

When describing his *DEVS* (Discrete Event System Specification) formalism, Zeigler (1984, 1990, 2000) proposed that a simulation model should be built in a hierarchical and modular fashion; i.e., a model is a collection of interconnected components, each of which deals with its own input, state transitions and output. These basic components can be combined to form "higher level" components, which can then be further connected and aggregated to construct a new sub-model. For building a complex model, this process can be repeated recursively. Such component architectures have since been used to develop many different types of simulators and other complex software systems or applications (e.g., see Alejandra, Mario, & Antonio, 2003; Atkinson, Bunse, Gross, & Peper, 2005). Some important concepts of component software development including methods for designing and composing them can be found in Jifeng, Li and Liu (2005).

Zeigler's DEVS formalism has bred two types of component technologies; those that focus on visual modelling such as the use of JavaBeans (Praehofer, Sametinger, & Stritzinger, 2001) and those that focus on distributed simulation environments such as CORBA (Yahiaoui, Hensen, & Soethout, 2004) and Microsoft's COM (Cho & Kim, 2002).

Visual modelling environments often organize components in a library (with its own internal logic) and offer a GUI for easy access to their properties and methods. Interfaces in which icons or blocks are attached to components and simulation structures can be quickly constructed via "drag and drop" interactions are often provided (Odhabi et al., 1998). Since the underlying library is typically based on an OOP metaphor, components support encapsulation, inheritance, polymorphism and exception handling. The advantages and disadvantages of such software architectures

have been discussed in detail elsewhere (e.g., Oses, Pidd, & Brooks, 2004; Valentine, Verbraeck, & Sol, 2003).

Each component is designed to guide messages' flows and to control their movements. Messages are generated by the first "upstream" components and then transferred to other "downstream" (listener) components; e.g., through output ports. Since downstream components are configured by upstream components (either at design time or during runtime), the only task of the downstream components is to react to messages they receive; e.g., by updating their own states, other components' states and/or the messages' states. To do this, they need no knowledge of where the messages have come from.

5.3 The Environment of Animated Simulation Models

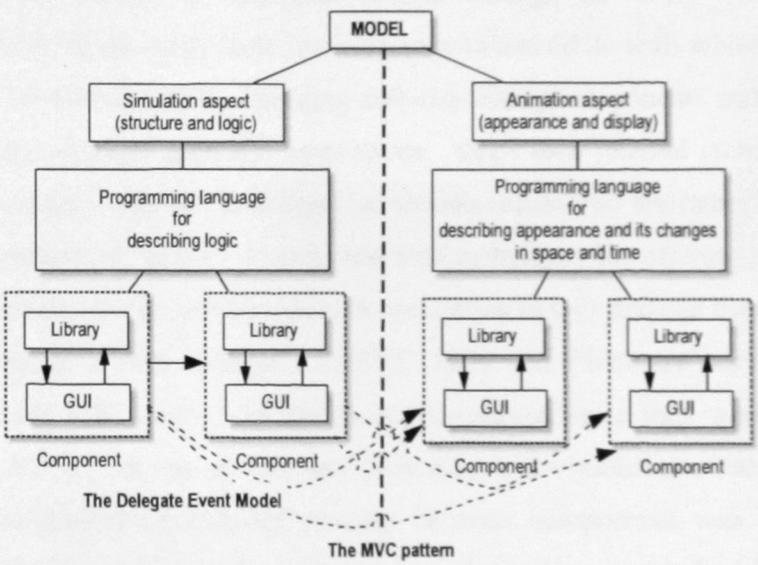


Figure 5.1 Simulation and Animation Aspects of a Model

Figure 5.1 shows the architecture of an animated simulation model. Note that we propose a clear separation between a model's *simulation* aspect (which describes model structures and logic) and its *animation* aspect (which traces model dynamics by showing the sequence of generated events and how its components' appearance and location will change over time). Although animation is optional (i.e., not all models need to be animated), it is an essential feature for observing and understanding

dynamic behaviour, verifying and validating models (Law & Kelton, 2000) and can prove particularly useful to generate *insights* rather than simply *predictions*.

As discussed earlier, the logic for a simulation model can easily be structured using a component approach. Recognizing the benefits offered by this approach, many component-based simulators have been built and reported; e.g., *XCELL+* (Conway & Maxwell, 1987), *SIMFACTORY* (Tumay, 1987), *simjava* (McNab & Howell, 1996, 1998), *JSIM* (John A. Miller, Youngfu Ge, & Junxin Tao, 1998), *Simkit* (Buss, 2000, 2002), *COST* (Chen & Szymanski, 2002), *JDEVS* (Filippi, Delhom, & Bernardi, 2002), *Viskit* (Buss & Blais, 2007) and *BPSim++* (Melão & Pidd, 2007). A common thread of all these tools is that they use input and output ports (either specifying through code or a GUI) to permit interactions between their components.

In term of ease-of-use, *Simkit* and *COST* are not user-friendly, since they only allow a model builder to construct models through an *API*. *XCELL+* and *SIMFACTORY*, on the other hand, provide easy-to-use GUIs with which simulation models can be constructed by dragging and connecting components and initializing their properties through graphical interactions. Since their internal architectures are hidden from users, however, these tools' extension capabilities are rather limited. To solve this problem, *BPSIM++* tries to combine techniques for offering both ease of use and flexibility, but its resulting models are written in C++ and can therefore not be accessed through a web browser. *JDEVS, JSIM* and *Viskit* are easy-to use and extensible tools with support for web-based simulation since they were developed using Java, but do not incorporate any animation and visualization facilities. The animation of displaying message passing between components was emerged in *simjava* but the visualization of model states was limited to text labels only which are placed over the components. Many modern simulation software, e.g., *Arena* (Kelton et al., 2004), *Flexim* (Nordgren, 2003), *SIMUL8* (Concannon et al., 2006) and *ProModel* (Harrell, Ghosh, & Bowden, 2004) meanwhile are excellent tools for building sophisticated DES models and analyzing system performances through animation and various visualization tools. However, their capabilities to support learning through user-directed experimentations during run time are rather limited.

5.4 The Delegation Event Model for Linking DES Components

The *Delegation Event Model* suggests a generic design for how to broadcast many different events (about which information is stored in an *event object*) from an *event source* to all registered *event listener* objects and invoke an appropriate method on them. This pattern offers flexibility since (1) a single event source can broadcast any number of events, (2) its listeners can register to receive any interesting events by just implementing interfaces that define the events, and (3) each listener can respond to a received event(s) in its own special way. To enable the *event source* class to broadcast many different events, it just needs to provide separate registration methods and listener lists for each class of event.

This style of event broadcasting is analogous to the flow of entities in DES components, where a temporary entity (an *event object*) is passed from an upstream component (an *event source*) to downstream components (the *event listeners*). Any downstream component can then act as an *event source* to further downstream components. Entities' and visited components' states will be updated during this process, which will continue until a message's path is completed and the message is removed. Thus, entities should have properties to store their current source component and target component; and optionally an array to store all their visited components.

The *Delegation Event Model* plays two important roles in building DES simulators. First, without implementing this pattern, model developers (e.g., teachers) must create a class which defines an entity type's *lifecycle* as discussed in Chapter 4. Writing such lifecycle descriptions become more complicated if entities need to be split (e.g., using conditional statements to represent probabilities and/or conditions) when they reach at a certain phase of their lifecycles. Second, through sub-classing, other tool designers can extend our existing architecture and create new high level components to support additional requirements (e.g., other simulation metaphors and styles). An example for these is a *record* component used to collect and report various types of observational statistics. Implementation of this would be easy, since a component can broadcast events to many interested listeners. Another example is a *renege* component that listens to a queue, removes relevant entities from the queue if their waiting time's tolerance threshold has been exceeded and then transfers the entities to certain locations.

Based on this pattern for tracing events triggered by message flows, DES components can be constructed to simulate and animate the transfer of many types of entities from one component to another, using the upstream components' output ports. We have used class and interface structures suggested by Moock (2004) to build a suitable implementation of DES components in Flash ActionScript, which is illustrated in Figure 5.2. DES classes in Chapter 4 will again be used for our discussion here. Note that these structures can easily be applied to implementations in other programming languages.

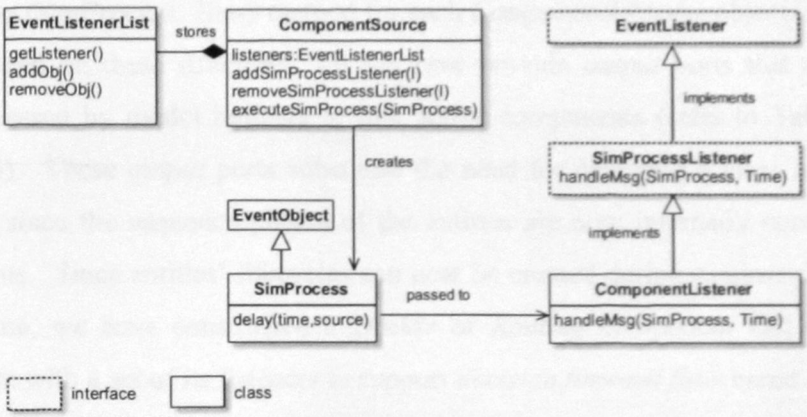


Figure 5.2 The DES Delegation Event Model Structure

We use five basic classes and two interfaces to implement DES components based on the *Delegation Event Model*; i.e., *ComponentSource*, *EventListenerList*, *EventObject*, *SimProcess*, *ComponentListener*, *EventListener* and *SimProcessListener*. The *ComponentSource* (an *event source*) represents classes that schedule an instance of the *SimProcesss* class (a *SimProcess* object) and broadcast this object to its registered listeners. Simulation specific *ComponentSource* classes include Sources, Queues, Servers, Sinks, etc. A *ComponentSource* object should be composed of *EventListenerList* objects; i.e., it should manage a list of the *ComponentSource*'s event listeners. The *ComponentSource* class can be equipped with a GUI to provide easy access points to its properties, including a point to specify its listener objects.

The *SimProcess* (an *event object*) class encodes entities that can be placed on an *Agenda* (a list to store the next scheduled event for a particular *SimProcess* object)

and will be broadcasted to *ComponentListener* objects when its scheduled event time is reached (i.e., when it should be activated by the simulation *Monitor*). The *SimProcess* class is derived from the *EventObject* class; a base class that holds a reference to the class that has scheduled it. In order to receive event notifications from a *ComponentSource* object, the *ComponentListener* class must implement the *SimProcessListener* interface; an interface that specifies a set of event methods.

The *SimProcessListener* interface implements the *EventListener* interface; a marker (empty) interface that enables event listener classes to be notified by *ComponentSource* objects. When an event occurs, the *ComponentSource* invokes a *handleMsg (SimProcess, Time)* method for each *ComponentListener* object.

Based on these structures, we can now provide *output* ports that should be easily accessed by model builders to link active components (refer to Table 4.7 in Chapter 4). These *output* ports substitute the need for declaring a class of *entities' lifecycles* since the sequence phases of the entities are now internally controlled by components. Since entities' lifecycles can now be created during runtime rather than design time, we have constructed a *Decide* or *Routing* component that couples a component with a set of its listeners to support *decision forward flow* based on certain control strategies; e.g., their types, probabilities, a shorter queue and server status.

Figure 5.3 traces a simple flow of a *SimProcess* object in an *M/M/1* queuing scenario. An instance of the *SimProcess* class (which contains data about its birth time, current phase, current location, etc.) is first created and scheduled in the *Event List* by invoking a *delay (time:Number, source:Component)* method on a Source component (which then becomes the highest upstream component). The *time* argument is the time that the next event for this *SimProcess* object is scheduled to occur and the *source* argument refers to the *ComponentSource* object that scheduled it. When the scheduled time comes, the *SimProcess* object is removed from the *Event List* by the *Monitor*. During the removal activity, the *SimProcess* object makes a call back to the event source that scheduled it (in this case a *Source* object) and invokes an *executeMsg (SimProcess)* method on the event source. This event source then executes relevant code (e.g., an animation method to move the *SimProcess* object to its downstream component or animate its physical appearance) and broadcasts the *SimProcess* object to its all registered listeners by invoking *handleMsg (SimProcess, Time)*.

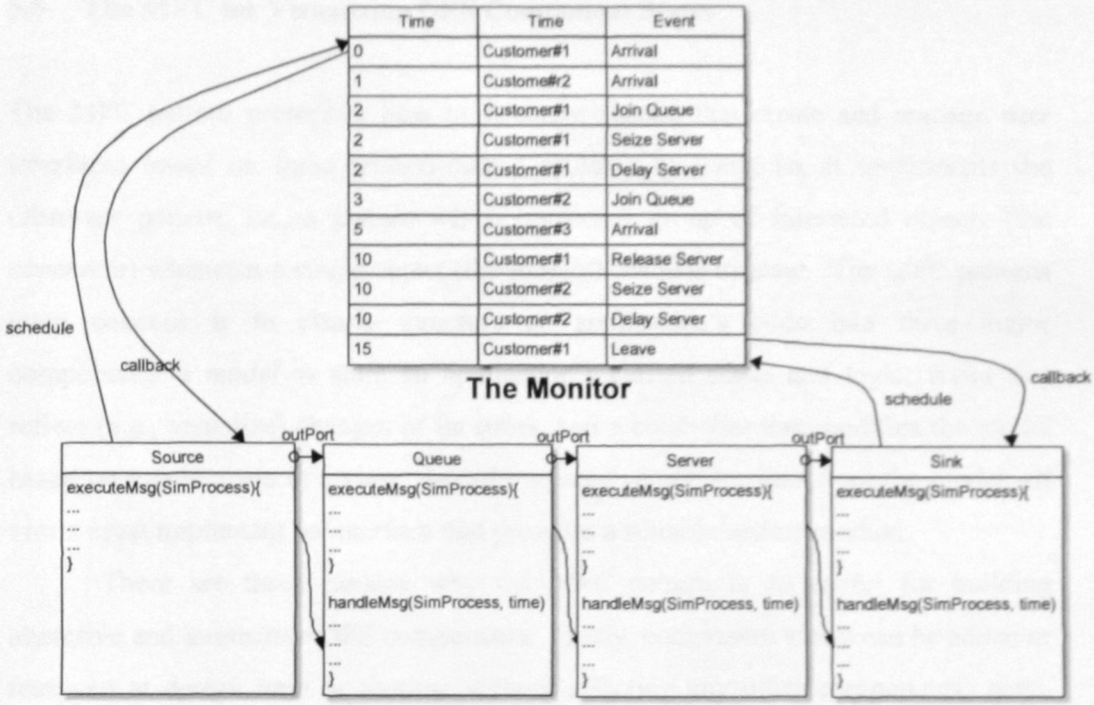


Figure 5.3 The flow of a *SimProcess* Object in DES Components

All registered listeners can respond to the *SimProcess* object in different ways, but one of them should instruct the *SimProcess* object to proceed to its next phase; i.e., by reinserting it into a suitable location in the *Event List*. When the next scheduled time is reached, the *SimProcess* object has to call the event source that scheduled it. The event source then executes *executeMsg (SimProcess)* and broadcasts the *SimProcess* object to all of its downstream components. This mechanism is repeated until the *SimProcess* object departs from the system; i.e., when it arrives at a *Sink*; i.e., its lowest downstream component.

Implementing the *Delegation Event Model* in DES classes not only enables us to link active components with each others, but it also allows us to control and simulate entities' delay time to their downstream components; i.e., to represent travel time from location to location. The travel time should then again be made accessible for modifications through the components' GUIs during design time and runtime. Permitting learners to change entities' travel time at any time they wish will help them to understand the effect of delay time to model performance.

5.5 The MVC for Visualizing DES Component States

The *MVC* pattern prescribes how to structure classes that create and manage user interfaces based on *input-process-output* cycles. In doing so, it implements the *Observer* pattern; i.e., a pattern which notifies a group of interested objects (the *observers*) whenever a single object (the *subject*) changes its state. The *MVC* patterns main concern is to clearly structure an application's code into three major components: a *model* to store an application's current states and logic, *views* that reflect (e.g., visualize) changes of its states, and a *controller* that modifies the model based on inputs made in a view. In order to receive notifications from the model, all views must implement an interface that provides a suitable *update* method.

There are three reasons why the *MVC* pattern is so useful for building attractive and interactive DES components. Firstly, component views can be added or removed at design time or runtime without affecting any other components' parts. Learners can therefore freely *customize* model visualizations. Secondly, all views are concurrently notified through an *info object*; i.e., an object that contains information about its subject's current states. This allows the synchronous display of *all* of a DES component's current states, either graphically (e.g., histograms, graphs, etc.) or in a more abstract fashion (e.g., texts, tables, etc.). Thirdly, when designed properly, many visualization tools (e.g., histograms, graphs, etc.) can be *reused* by different types of DES components (e.g., sources, servers, etc.).

Figure 5.4 shows generic *MVC* implementation structures for a single DES component. This involves seven basic classes and four interfaces that cooperate with each other to provide a GUI and suitable visualizations. The *ComponentModel* (e.g., sources, queues, servers, sinks, etc.) class broadcasts its states to all registered observers through its *ComponentUpdate* object (info object). This is an object that stores its current states. Each *ComponentModel* class should have its own *ComponentUpdate* class with a unique name (e.g., *SourceUpdate, QueueUpdate, ServerUpdate*, *SinkUpdate*, etc.).

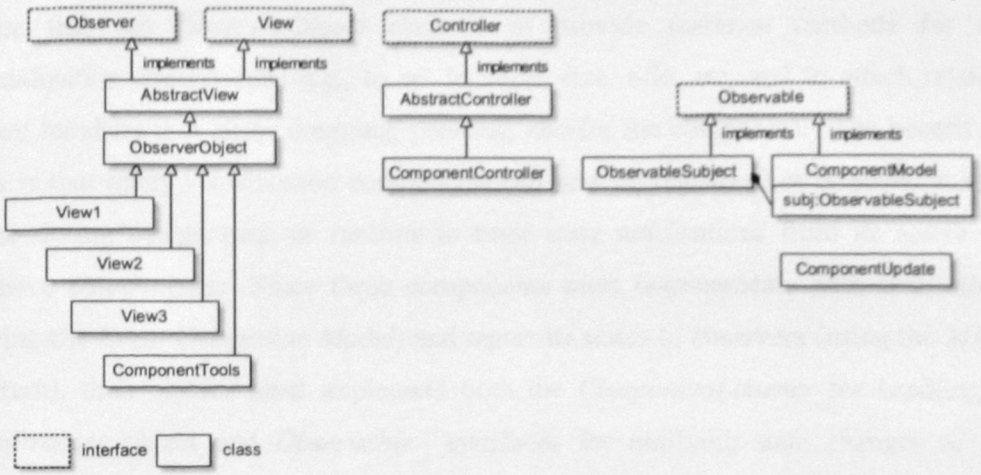


Figure 5.4 The DES *MVC* Structure

The *ComponentModel* class implements the *Observable* interface to provide abstract methods for maintaining and notifying *Observer* objects. The implementation for the *Observable* interface is provided by the *ObservableSubject* class. An instance of the *ObservableSubject* class is used in the *ComponentModel* to broadcast updates to its observers whenever its internal states have changed. By implementing the *Observable* interface, the *ComponentModel* class can freely inherit from any other class; i.e., it can be a subclass of other class.

To receive input from its views, each *ComponentModel* class must have its own controller (e.g., *SourceController*, *QueueController*, *ServerController*, *SinkController*, etc.). The model's controller must extend the *AbstractController* class; a class that provides basic services specified in the *Controller* interface. The *Controller* interface in turn contains references to the model and its view. To receive notifications about state changes in the *ComponentModel*, all interested views must extend the *AbstractView* class; a generic implementation of the *View* and *Observer* interfaces. The *View* interface contains abstract methods to set and retrieve the model and controller objects observed by this view, while the *Observer* interface contains an abstract *update( )* method. It is up to a view's update method to react to the information object sent by a *ComponentModel*.

We can now make some modifications so that the visualization components (e.g., *Clocks*, *Histograms*, *Graphs*, *BoxPlots*, *Levels* and *Tables*) to be derived-classes (subclasses) of the *AbstractView* class; i.e., the class that extends the *MovieClip* class.

Note that the *ObserverObject* class is to provide common methods for all visualization components; e.g., to set location, size, title, etc. and to attach related event handlers that allow dragging, pressing, etc. for the component. The benefit of this is that many visualization components can now be registered or removed at any time during design time or runtime to trace state notifications from its active or passive components. Since these components must communicate with each other (using the *Event Delegation Model*) and report its states to observers (using the *MVC* pattern), their classes must implement both the *ComponentListener* for handling a *SimProcess* object and *Observable* interfaces for notifying state changes to its observers. Note that a visualization instance only receives the notification of its active or passive component states from the time point it is created. This could offer some benefits; e.g., learners can inspect in detail the performance of the model and compare its performance from various simulation times. To receive the notification at simulation time zero, learners must create all interested instances before running the model.

Implementing both design patterns in a DES component permits a loose coupling among DES components and its visualization components. Because of this flexibility, we have created a utility component called *visualization palette* that floats on the top of a model during runtime and holds various types of visualization tools to allow learners to customize the model's GUIs. Various model GUIs can be created by instantiating a new visualization instance (i.e., clicking its symbol on the palette), registering it to receive the notification from a relevant component's state changes (i.e., dragging a point on it and dropping the point onto the component) and dragging it to any location on the stage. However, since these processes demand some efforts from learners and not all visualization tools can be associated to a component (e.g., a *Clock* component can only be used with the *Monitor* component), this approach is not so effective for a learning environment.

To overcome this problem, we directly embedded a list of visualization tools on the components' GUIs. Learners only need to click a command button (each of which associates to a new type of visualization tools) to instantiate a new visualization tool. We believe this approach will help them to understand the dynamic behaviour of a DES model.

5.6 Connecting External Data

Allowing learners to save the current states of a model offers some benefits in learning and teaching. These include permitting them to retain the model's visualization and parameter settings and mark time points of interesting scenarios. Unfortunately, this feature is not offered by existing DES tools. As a result, learners are always presented with a new fresh model each time it is loaded.

Saving a DES model requires us to store model relevant structures and states to a file. Generally, there are three types of files for storing application data: text files, databases (Rob & Semaan, 2000) and XML (Hunter et al., 2000). These files will be accessed to reflect the current behaviour of an application and can be updated to save the application's latest information during running time.

Text files are supported by many applications, easy to create and use and readable by humans. However, they cannot store complex data structures as in DES models since information storing is restricted in a sequence of lines (i.e., a list of name-value pairs). Databases ease an application to access data through the use of query languages. They have been used for storing DES static structures as implemented in *Arena* software. However, designing, creating and linking dynamic tables that store DES temporary entities and data fields for updating (storing or deleting) timely changed DES model components (especially visualization tools) is unpractical.

XML provides a good data storage for DES models due to its ability to support complex data structures for storing entities and components with their own properties. Additionally, the current structures can easily be extended to support additional levels of more complex DES data structures. However, the process of creating and updating these structures can only be done in the server for a security reason. As a result, XML is usually used for storing and accessing data than updating the data, unless the updating process is done manually (Castillo et al., 2004).

To eliminate these constraints, Flash has introduced *Local Shared Objects* (LSOs) that store an application's relevant information (especially its parameter settings) on users' computers. Thus, each time they access the application through their computer, they will get the updated version of the application. This makes the application looks like it has been customized for each user.

The main advantage of LSOs is that data can be stored in various data types (e.g., number, array, boolean, date, XML objects, etc.), making the storing processes of various objects are quite straight forward. However, little Flash interactive movies have exploited its potential since it is usually used for storing basic data; e.g., user names. For this reason, we used LSOs for storing our DES models' states, animation and visualization instances. The ideas behind this implementation can easily be applied in XML with little effort.

Each DES component and entity should have its own LSO file (with a ".SOL" extension) and be named based on its instance name on the Flash stage. The main storage location for LSO files is operating system-dependent but it is typically located under the *Flash Player\#SharedObjects* folder. All LSO files belonging to a DES model are saved under a subfolder (under the main storage location) named based on its DES model file name to avoid conflicts with other models' LSO files. We thus need to retrieve the DES file name using *ActionScript* code whenever the model is reloaded. Since the LSO name exactly follows its object name, entities (i.e., *SimProcess* objects) and visualization instances that are created during runtime must be coded so that each of these objects has their own unique names. However, Flash will automatically assign a default unique name for an unspecified object name. Thus, the issue of an object without a name will not arise.

We created a *Utility* component as a means to save component instances and their states. It has a *Save* button for instructing all objects (in the form of *MovieClips*) on the Flash stage to detect the existence of their associated LSO files. This can be of two cases.

If their LSO files have not existed (i.e., the model has not been saved, or new *SimProcess* or visualization instances have been created since the last save), we need to command the objects to create their LSO files and store their relevant property values. In case of active or passive components, we can directly transfer information in their info objects to their LSO files.

If the LSO files exist (i.e., the model has been saved before), we only need to update these LSO files with their latest property values. Note that the updating processes will only take place at the points where learners opt to save or resave the model, not during the whole process of model running. This is to ensure that information in the LSO files is preserved until the next saving point so that learners will only be presented with a model of the latest saving point. The *Utility* component

has other buttons; the first one is to flush all LSO files for a model, i.e., to get a fresh model with its default values and the second one is to show all the paths of entity movement for clarifying the sequence of events in the model. The paths are presented by arrows that link active or passive components based on their output port parameters.

Supporting such a saving capability needs all components to have certain features. First, each active and passive component needs to transfer the current list of its observers (we have had an array for this since we implement the *MVC* pattern) into its own LSO file and consequently instruct all these observers to create (or update) and store relevant information in their LSO files every time the model is saved. Second, a *Source* component needs to have an array for holding a current list of its created *SimProcess* objects that are still available on the stage at certain points of time. Note that we do not have this in our previous *Source* components. This array needs to be updated each time a *SimProcess* object is created or destroyed (i.e., all *SimProcess* objects will remain in the list until they are destroyed by a *Sink* component).

If learners opt to save the DES model, the current list must be transferred into its *Source*'s LSO file. Sequentially, each of the *SimProcess* objects is to create its own LSO file (or update if its LSO file has existed) to store their current information; e.g., their latest locations, birth times, left time to finish a certain activity, etc. The *Source* component also needs a variable to store the latest number of generated entities so that it can extent this number when the model is re-run. Third, all scheduled events in the *Monitor* (i.e., events that have not been cancelled in the *Agenda*) need to be transferred to respective *SimProcess*'s LSO files whenever learners save the model. Thus, we have to make sure that the *SimProcess*'s LSO files have already existed before transferring a list of their unexecuted events (with their time of occurrence) to their LSO files.

Whenever a model is loaded or refreshed (after saving the model using the *save* button in a *Utility* component) in a web-browser, a *Source* component will first get its current list of *SimProcess* objects from its associated LSO file and then create those entities. Each time a *SimProcess* object is created, all scheduled events stored in its LSO file will be retrieved and inserted to the model's *Agenda*. Consequently, each active and passive component reads its LSO file to initialize its parameter settings and creates visualization instances based on its list of observers. Each

visualization instance will then be matched with its LSO file and fed with the data stored in the file. Through these processes, learners will obtain the model with the previous animation, visualization and component parameter settings.

A tricky issue arose when we wanted to resave a model; i.e., the model that has previously been saved is loaded and re-run. During this point onward, some objects (e.g., entities that have left the model or certain visualization instances that have been removed by learners) have to be destroyed to save computer memory. If we automatically destroy the LSO files along with their associated objects and learners opt to discard any changes during this time interval, we will lose the LSO files. As a result, if the model is re-loaded, some objects will be reinitialized with their default values due to the missing of their LSO files. However, if we just destroy the objects (i.e., we do not automatically destroy their LSO files) and learners opt to resave the models, we will keep a number of worthless LSO files; i.e., a list of orphan LSO files without their owners. This is particularly true for a model that contains many active entities and/or has been extensively experimented with various visualization tools.

To solve this problem, we programmed *SimProcess* objects so that they destroy themselves when they exit a model but their associated LSO files are still available until a certain point of time. For this, the *SimProcess* objects should communicate with its creator; i.e. the *Source* instance that creates them. To do this, the *Source* instance temporarily stores a list of destroyed objects. If learners want to resave the model, this list will destroy all stored objects' associated files, else nothing will happen. The same thing applies to any removed visualization instances where each active component needs a temporary array to store its removed observers, and then removes the relevant visualization instance' LSO file in case learners opt to save the model.

We also need to maintain the smoothness of animation whenever a model that has previously been saved is loaded to be run for the first time. At any saving point, the model is bound to have some entities that have not completed movement to their destinations. These entities can be at any path; each of which has used some amount of its route time to reach its destination. Anytime we load and re-run the model, we have to ensure that each entity continues its movement from the previous stopping location to its destination using only the remaining time left.

We handle these entities' residual movements by delaying the execution of the model's *Event List*. Using this approach, the entity that has the smallest remaining time can finish its movement based on its residual time. Other entities meanwhile use this time to step toward their destinations before the *Monitor* executes the next scheduled event and updates model time. Without delaying the execution of the *Monitor*, the entity will jump directly to its destination while other entities remain static in their previous locations until their scheduled times have been reached that denote the times for them to jump to their destinations. Since we move *SimProcess* objects based on movement steps (the partition of these steps depends on its time delay and a viewing ratio; refer to Chapter 4), delaying the *Monitor* only needs us to store the number of remaining steps left to reach the destination in the entity's LSO file.

5.7 Example

This section discusses the ease of use aspects of our components in building queuing networks and how final queuing models allow learners to conduct various experiments and visualize model behaviour through their GUIs. Figure 5.5 shows a snapshot of the DES components and their locations within the Flash environment.

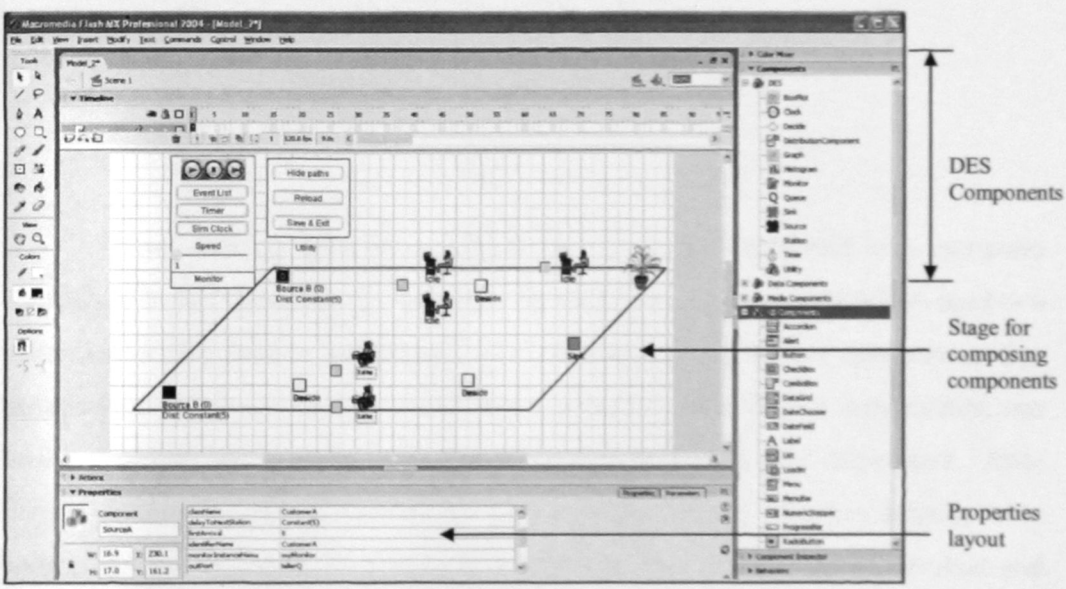


Figure 5.5 Flash Development Environment

All components reside in the Flash's *Components* panel and can be instantiated by dragging them onto the Flash's *Stage* to construct any types of queuing networks; i.e., open networks, closed networks or mixed networks (see Bose, 2002; Gelenbe & Pujolle, 1998). The construction of these networks is accomplished by utilizing *Decide* component instances that route entities to their downstream components based on three options: probabilities, a shortest queue or entity types.

To demonstrate the ease of use of our DES components, we will develop a sample of a queuing network as illustrated in Figure 5.6. This sample simulates two types of entities arriving into a system. The first type joints a single queue and will then be served if one of the two available servers is idle. Upon completion, these entities need to go to another queue before leaving the system. The second type chooses the shortest queue between the two available queues. After being served, some percentage of the entities exits the system while others need to go to the servers that process the first type of entities. They are then free to leave the system.

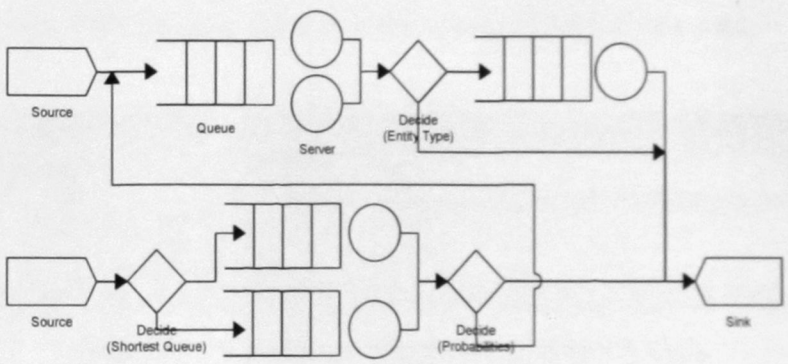


Figure 5.6 A Queuing Network System

These queuing network structures can easily be transferred to a computer simulation model using our components. Based on these structures, teachers need two instances of the *Source* component, four instances of the *Queue* component, five instances of the *Resource* component, three instances of the *Decide* components, one instance of the *Sink* component and one instance of the *Monitor* component. Note that a *Monitor* instance is needed by all simulation models. Its functionality is to coordinate the sequence of entities in a model so that entities can be invoked and transferred between components at appropriate times and in the right orders.

All of these component instances need to be dragged and dropped onto Flash's *Stage*. Once they are on the *Stage*, teachers can arrange the component instances' locations accordingly, give them a name and access their properties through the *Properties layout* panel (see Figure 5.7). The process of dragging, dropping, naming an instance, initializing its parameter values and specifying its targeted components is repeated until the simulation model structure has been constructed.

All components must have unique names to correctly link them with each other; i.e., these names are specified in their upstream components' output port properties so that these upstream components can route entities to their downstream components. This approach avoids us from writing case statements to represent the entities' lifecycles as in our example in Chapter 4. All components have their default property values that specify their behaviour during runtime and can be changed by clicking the appropriate row in the *Properties layout* panel. For example, a *Server* instance has properties as listed in Table 5.1. Once the simulation structure has been built, other visualization tools can then dragged, dropped at appropriate locations and connected to the DES components to provide a default GUI for the model.

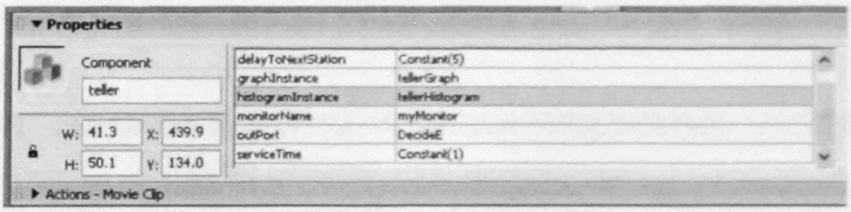


Figure 5.7 A Server's Properties and Default Values

Table 5.1 Server Properties and Description

Properties	Description
capacity	Number of resources that can be seized by entities in a queue
delayToNextStation	Time taken (based on a distribution type; e.g., Constant, Exponential, etc.) for entities to reach the next component
graphInstance	Name of a graph instance to display capacity used vs. simulation time
histogramInstance	Name of a histogram instance to display service times
monitorName	Name of a monitor instance that sequences state transitions of all types of entities in a model
outPort	Name of the next component to transfer entities
serviceTime	Type of distribution specifying processing time

Figure 5.8 shows a sample of the final model constructed in this manner with its own customized visualizations. The model allows learners to stop, increase and decrease the animation speed for their best visualization effect (Figure 5.9a), conduct various experiments through an interactive GUI and observe the impact of changes to model behaviour through a range of engaging visualizations. Conducting experiments are easy since they can change any component's parameters at any time they wish (i.e., by clicking the component and typing appropriate values into text boxes and/or choosing one of several options in combo boxes) and directly visualize the component's internal states by clicking available command buttons. For example, learners can change priority rules (queuing disciplines) for queues (Figure 5.9b), alter the distribution of time between arrivals for the two types of entities, modify capacities and service times for servers (Figure 5.9c) and interact with data visualizations; e.g., changing minimum and maximum values, and the number of intervals of histograms (Figure 5.9d). The ability to change histogram parameters enables learners to view the distribution of data in a variety of formats. Labels of important components' current parameter values are also displayed during runtime for model clarification.

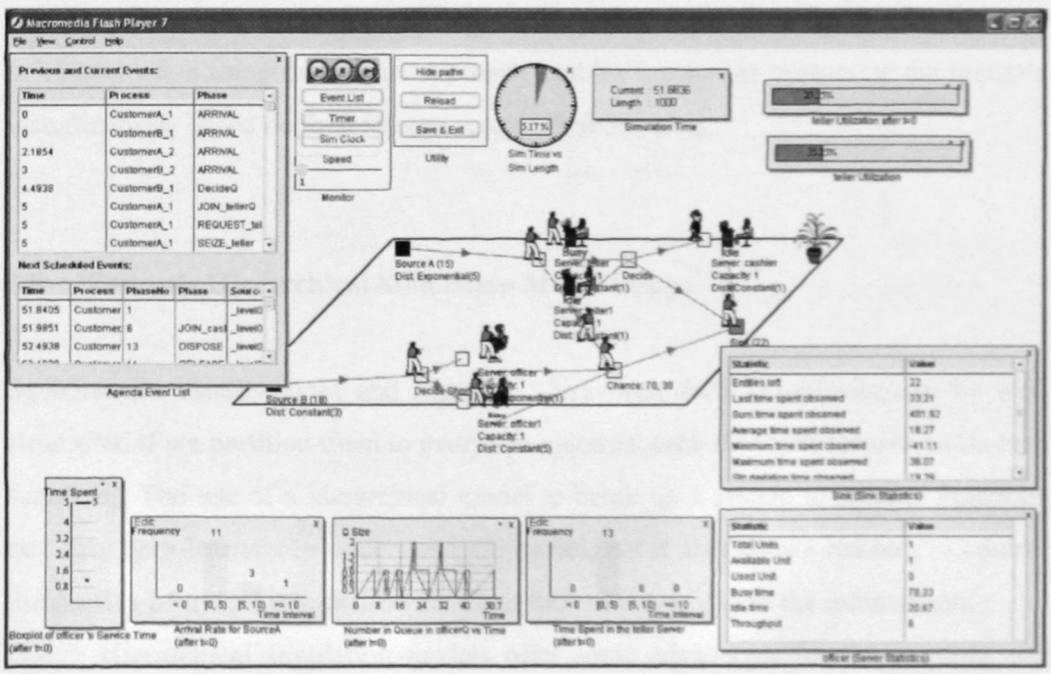
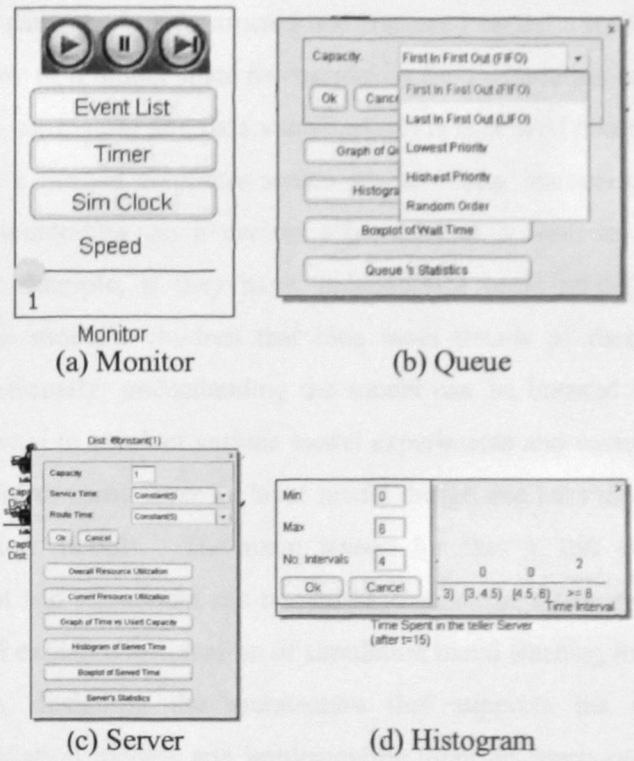


Figure 5.8 A Final Model



(a) Monitor (b) Queue

(c) Server (d) Histogram

Figure 5.9 Interactions with Component Instances

All data visualization (that reports the model's performance during the simulation run) selected by learners can be located at any location on the model stage or closed when unneeded. This approach enables learners to customize the model's visualizations based on their interest to ease their learning.

5.8 Towards Hierarchical Simulation Model Design

Systems are usually large and complex. However, their complexity can be well structured if we partition them to many sub-systems; each of which focuses on its own function. The use of a hierarchical model to break up a system to smaller functions not only help learners to understand the model, but it also allows learners to control the display of model information based on their ability to digest the information.

Hierarchical simulation models offer some advantages for the learning and teaching environment. First, teachers can structure a large and complex simulation model to different layers of abstraction; i.e., by building and representing the model

from a basic, general model to more detail sub-models (its child models). Thus, a complex model can now be constructed and managed easily. Second, learners can have a better view of a model since its complexity (i.e., simulation components, their interconnections, animation and data visualization) is now well controlled to limit its crowdedness on a limited computer screen space. Thus, learners can control their learning by concentrating on a certain sub-model at a time in which they are interested. For example, if they have understood a basic model, they can now transverse to the model's children that hide more details of their structures and functions. Additionally, understanding the model can be boosted if at each layer, learners are allowed to conduct various model experiments and customize the layer's visualization. Third, using layer by layer model design can ease the development of various simulation models. The main reason for this is that each component, visualization tool and sub-model can be reused to construct a new type of simulation model. This will expedite the creation of simulation based learning materials.

However, designing the architecture that supports the development of hierarchical simulation models and implementing them on computer will post some challenges. These include:

1) How to connect and synchronize a model with its children in a hierarchical fashion since parent models are dependent on their child model(s). This requires us to design a mechanism not only to synchronize the flow of entities in a relevant layer but also to properly transfer these entities to its child model and back to the layer whenever the entities exit the last components of the child model.
2) How to hide and display animation and visualization of sub-models at an appropriate time so that the model abstraction can be controlled properly.
3) How to store model states, animation, learners' experiment parameters and their customized visualization for each model layer so that when they revisit the layer, they will get back the settings they have had before.

Figure 5.10 shows an example of a hierarchical construction of a DES model. The model is partitioned to four layers (*Layer 1* to *Layer 4*). The execution of a particular layer depends on other layers. The top layer (i.e., *Layer 1*) represents the overall function of the model while the lower layers give more information about their

upper layers' functions. Each layer except the lowest layer has a sub-model symbol that hides its structures (components and their connections) that perform its function. Clicking this sub-model symbol will take learners to a lower layer (i.e., the layer's structures) while hiding the layer (e.g., through a button or a menu) will bring learners back to its upper layer. At any layer, there could be a sub-model that generates and handles their own type of entities, but these entities will not be transferred to any other layers. The flow of these entities must also be synchronized with the whole model time.

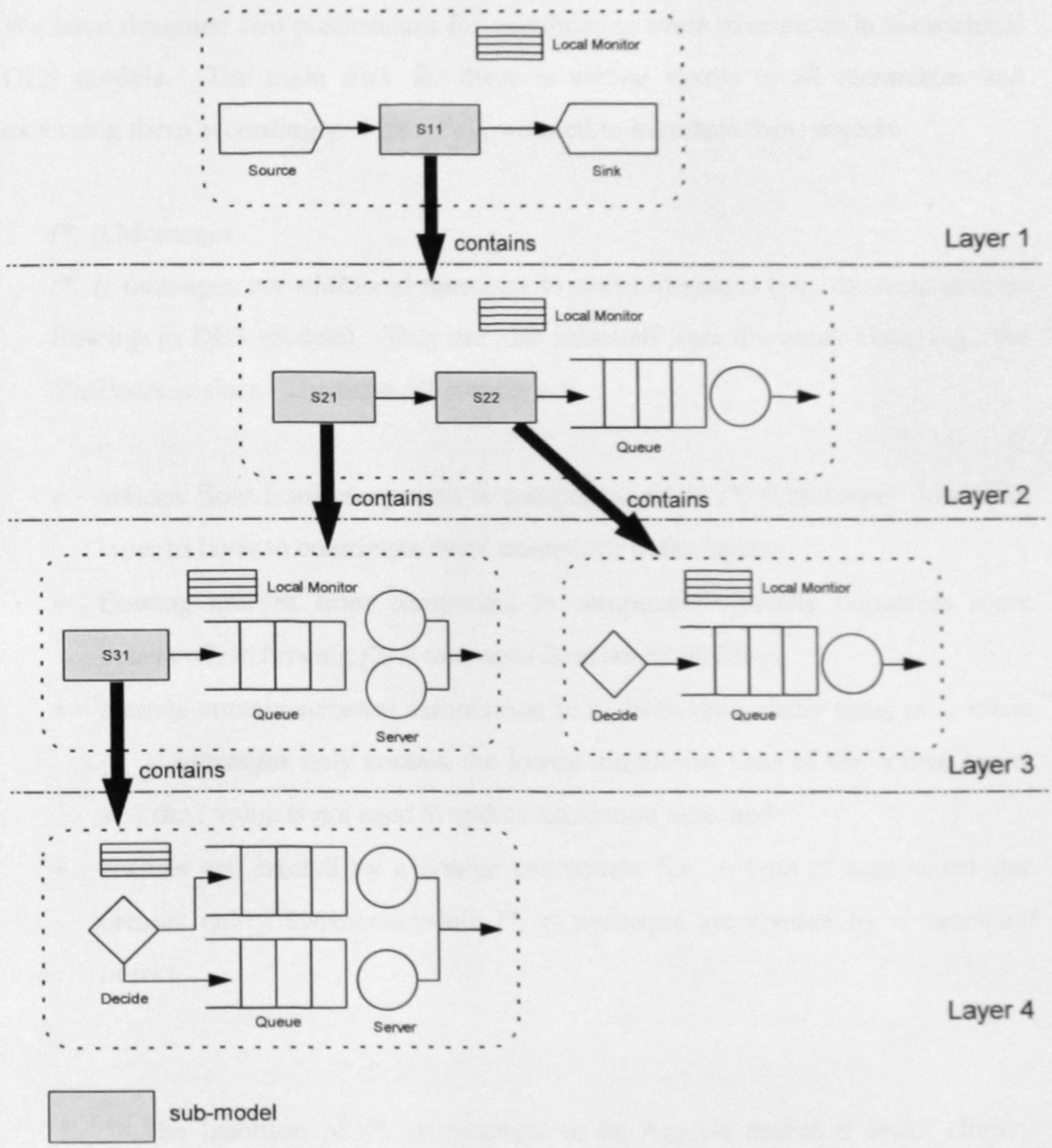


Figure 5.10 Hierarchical Construction of a DES Model

Each layer has its own window for locating its component structures and supporting its animation and visualization development. Entities that flow on this window must be well synchronized with its lower layers; i.e., entities should appear at a sub-model symbol at the right time once they exit their lower layer based on their time delays.

5.9 Designing Mechanisms for Hierarchical DES Models

We have designed two mechanisms for coordinating event executions in hierarchical DES models. The main trick for these is sorting events in all hierarchies and executing them accordingly. First of all, we need to introduce these objects:

1. *(*, t)* Messages

   (, t)* messages are additional messages to entity messages (i.e., dynamic entities flowing in DES models). They are also inherited from the entity class; e.g., the *SimProcess* class. The main differences are:

   - entities flow from component to component while *(*, t)* messages flow from layer to layer to coordinate event executions in the layers,
   - flowing entities from component to component typically consumes some delays while flowing *(*, t)* messages does not incur delay,
   - entities contain personal information (e.g., birth time, delay time, etc.) while *(*, t)* messages only contain the lowest simulation time of the source layers and the *t* value is not used to update simulation time, and
   - entities are created by a *Source* component (i.e., a type of component that creates entity instances) while *(*, t)* messages are created by a *Submodel* object.

   The insertion of *(*, t)* messages to an A*genda* makes it looks clumsy. However, their existence is important to tally all event executions.

2. *Submodel* Objects

A *Submodel* object encloses another layer. Entities arriving at a *Submodel* object could be in one of two cases: (1) the entities are from the same layer's previous component, or (2) the entities are from a lower layer's last component; see Figure 5.11. To differentiate these entities, the entity class needs to have a property; e.g., named *fromLayer* that takes a value of *current* (the first case) or *child* (the second case).

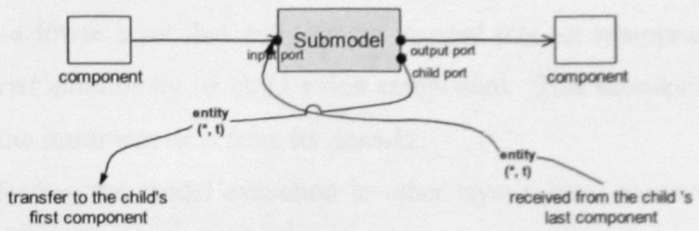


Figure 5.11 *Submodel* Architecture and Transferring Mechanisms

For the first case, the entities continue their flows to a lower layer's first component through a *child* port; i.e., a port specifying the child model's first component. For the second case, the entities flow to the same layer's next component through an *output* port; i.e., a port storing its downstream component.

3. *Local monitor*

Each layer has its own *local monitor* that executes the layer's activities stored in its *Agenda* in the right order.

5.9.1 Monitor Delegation Mechanism

When a model is loaded, each *Submodel* inserts a *(*, t)* message to its *local monitor*. This is to find the layer that has the lowest simulation time; e.g., in case of a *Submodel* object contains its own types of entities, or a *Submodel* object is the first component that locates a *Source* component under it. The model execution starts with the top layer's monitor removes the *(*, t)* message and transfer it to its lower layer's first component which then inserts the message to its *local monitor*. This process

continues until the imminent entity is found in a relevant layer. The entity will then be executed so that it can flow to the same layer or to another layer. Their flows to another layer must be accompanied by a *(*, t)* message.

The imminent item after this first iteration can be of two types: *(*, t)* object or entity object. If it is a *(*, t)* object, the execution of the current *local monitor* is passed to either its lower or upper layer' monitor depending on the source of the *(*, t)* message. Otherwise, it is flowed to the next destination; i.e., a component or a *Submodel* object. For a *Submodel* object, the entity with a *(*, t)* message is transferred to a lower layer that will then be inserted into an appropriate location in the layer's *local monitor* by its child's first component. This monitor then executes and removes the imminent item from its *Agenda*.

Transferring the model execution to other layer's *local monitor* implies that the layer contains lower next schedule time compared to the previous layer. The execution of this current layer's *local monitor* continues until another *(*, t)* message is found in its *Agenda*. These processes are illustrated in Figure 5.12. Figure 5.13 and Figure 5.14 meanwhile show some code under the *handleMsg(SimProcess, time)* and *executeMsg(SimProcess)* methods for the *Submodel* class and the simulation component class.

Basically, the *Monitor Delegation Mechanism* coordinates the execution of events in a hierarchical DES model through these mechanisms:

1. Instruct *Submodel* objects to insert *(*, t)* to each *local monitor*. Execute the top layer's monitor, followed by other layers.
2. Determine the imminent item type and the component that executes it.
3. (a) Flow the item to its next component in the same layer if the item is the type of entity and the component that executes it is a simulation component, or
   (b) Transfer the item and a *(*, t)* message if the item is the type of entity and the component that executes it is a *Submodel* object; see *Layer 1* in Figure 5.12. Insert them at appropriate locations in the layer's *local monitor*. This process should be done by the child's first component upon receiving the messages. Transfer the model execution to the layer's monitor.
4. Retrieve and remove the next imminent item from the current layer's *local monitor*. If the item is the type of *(*, t)* message, transfer the monitor execution to

the layer where the *(*, t)* is from and then repeat this step 4. Else, repeat the step 2.

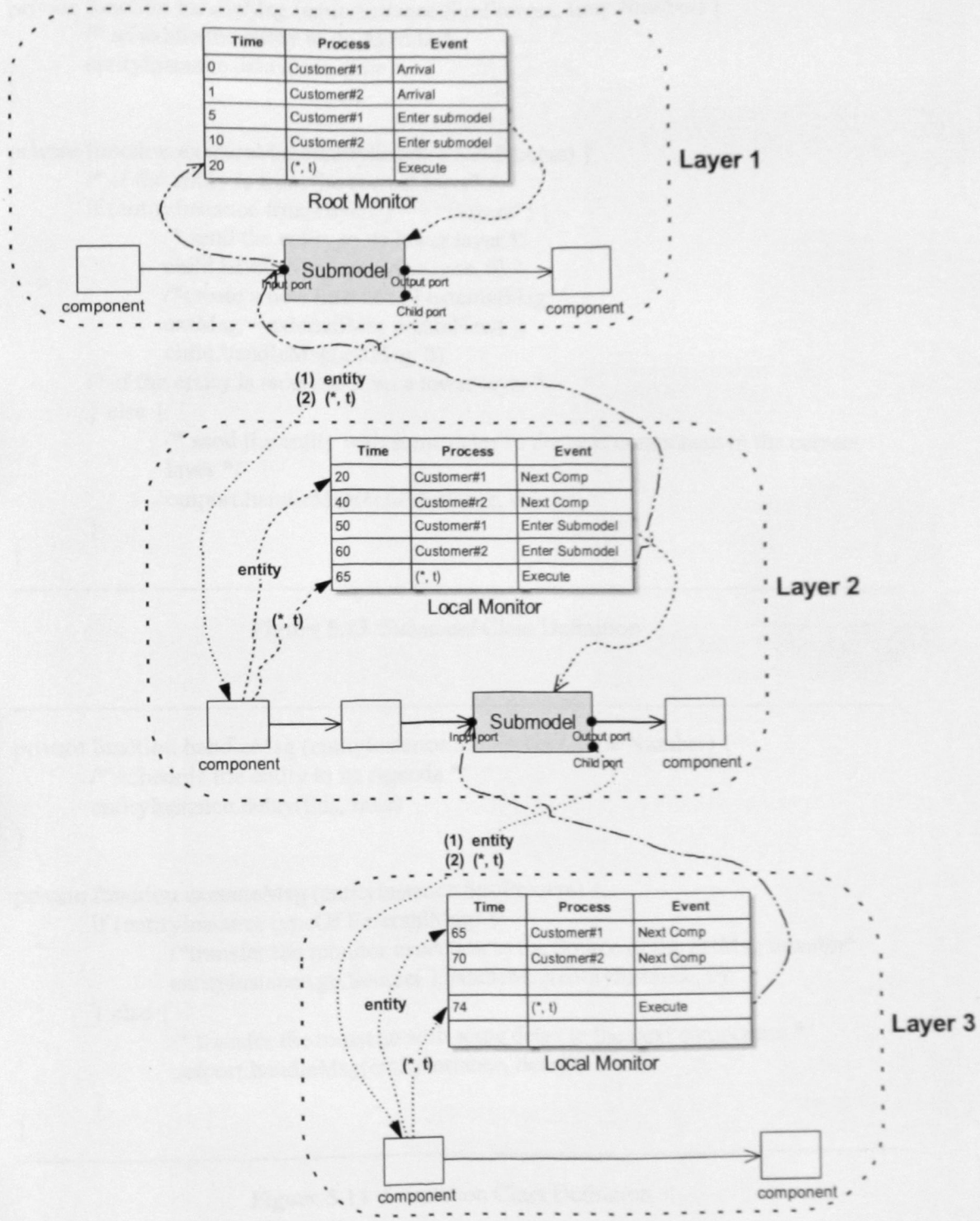


Figure 5.12 Monitor Delegation Mechanism

```
private function handleMsg (entityInstance:SimProcess, time:Number) {
        /* schedule the entity to its Agenda */
        entityInstance.delay(this, time)
}

private function executeMsg (entityInstance:SimProcess) {
        /* if the entity is from the current layer* /
        if (entityInstance.fromLayer( ) = = "current") {
                /* send the entity to its lower layer */
                child.handleMsg(entityInstance, 0)
                /*create a new instance of externalMsg*/
                extMsg = externalMsg.createNew( );
                child.handleMsg(extMsg, 0)
        /* if the entity is received from a lower layer */
        } else {
                /* send the entity with some delay to the next component in the current
                layer */
                outport.handleMsg(entityInstance, delay);
        }
}
```

Figure 5.13 *Submodel* Class Definition

```
private function handleMsg (entityInstance:SimProcess, time:Number) {
        /* schedule the entity to its Agenda */
        entityInstance.delay(this, time)
}

private function executeMsg (entityInstance:SimProcess) {
        if (entityInstance typeOf ExternalMsg) {
                /*transfer the monitor execution to the Source of the extMsg monitor*/
                entityInstance.getSource( ).handleMsg(entityInstance, 0);
        } else {
                /* transfer the message with some delay to the next component */
                outport.handleMsg(entityInstance, delay);
        }
}
```

Figure 5.14 Simulation Class Definition

5.9.2 Monitor Communication Mechanism

The *Monitor Communication Mechanism* differs from the *Monitor Delegation Mechanism* in two ways. First, *(*, t)* messages are sent by a monitor, not by a *Submodel*. However, a *Submodel* object and the last simulation component in a layer still transfer entities (i.e., *SimProcess* objects) to its lower and upper layer respectively. Second, for each iteration, monitors located above the source of a *(*, t)* message must all be executed sequentially rather than transferring monitor execution to a relevant layer. Such monitor communications through broadcasting *(*, t)* messages demand the monitor to implement the *Delegate Event Model*.

The purpose of broadcasting *(*, t)* messages down to a certain layer where the *(*, t)* comes from is to find the model's lowest simulation time in all visited layers' *Agendas*. For this, two types of iterations are needed. The first iteration broadcasts a *(*, t)* message from the top layer until the lowest layer to consider the cases of *Source* components are located in the lowest layer or certain layers have their own types of entities. The second iteration onward only involves broadcasting a *(*, t)* message until a relevant layer since any lowest next scheduled time below this layer definitely has a bigger value. This can be achieved by detecting the origin of a *(*, t_n)* message.

The *(*, t_n)* message is actually a *(*, t)* message containing the latest value of the lowest next scheduled time. This value is collected during its traversal to the top layer. By broadcasting the *(*, t_n)* message up from layer to layer, a parent layer acknowledges its child layer's lowest next scheduled time. For example, *Layer 1* stores the lowest next scheduled time for *Layer 2*; *Layer 2* stores the lowest schedule time for the *Layer 3* and so on. Thus, the execution of the child layer is controlled by its parent monitor. The details of the *Monitor Communication Mechanism* are as follows:

1. Insert a default *(*, t)* message in the *root Agenda* whenever the model is first run.
2. Broadcast the *(*, t)* message from monitor to monitor in a sequence order (*Layer 1*, *Layer 2*, *Layer 3*, ...) until it reaches the lowest monitor.
3. Execute the local monitor to coordinate events in the layer each time the layer receives the *(*, t)* message. For example, execute the local monitor in the *Layer 2*, followed by the *Layer 3* and so on. Consequently, send the *(*, t)* message to lower monitors.

4. Once the *(*, t)* message reaches the lowest layer's local monitor, retrieve the imminent item in its *Agenda*. Take its lowest scheduled time. Update the *(*, t)* message with a *(*, t_n)*, where t_n is the lowest next scheduled time for the layer. Broadcast the *(*, t_n)* to its parent monitor; i.e., the local monitor in its upper layer. Note that the *(*, t_n)* message is supposed to traverse up to the top layer.
5. Once the *(*, t_n)* reaches its upper layer's local monitor, insert the message at an appropriate location in its *Agenda* based on the t_n value. Retrieve the imminent item from the *Agenda*. Broadcast a new *(*, t_n)* message (could be the previous *(*, t_n)* message if it is the imminent item) to its upper local monitor. Repeat these processes until the *(*, t_n)* reaches the top layer. This will guarantee that each layer stores its child's lowest next scheduled time.
6. Once the *(*, t_n)* reaches and has been inserted to the top layer's *Agenda* (i.e., *root Agenda*), execute the *root* monitor. If the imminent item in its *Agenda* is the type of *(*, t_n)*, send another *(*, t)* message down to the layer where the *(*, t_n)* message is from. During this traversal, execute all visited layers' *Agendas* to remove the *(*, t_n)* messages. Note that only the layer that has generated the *(*, t_n)* message will create a new event (i.e., flowing a relevant entity); other layers only remove the message from their *Agendas*. Broadcast another *(*, t_n)* message. Repeat step 5.
7. Stop the processes if the length of simulation time has been reached.

Figure 5.15 traces a sample of *Agendas* based on the *Monitor Communication Mechanism*. The figure is split up to (a), (b) and (c); each one shows the *Agendas* at simulation time 0, 10 and 14 respectively.

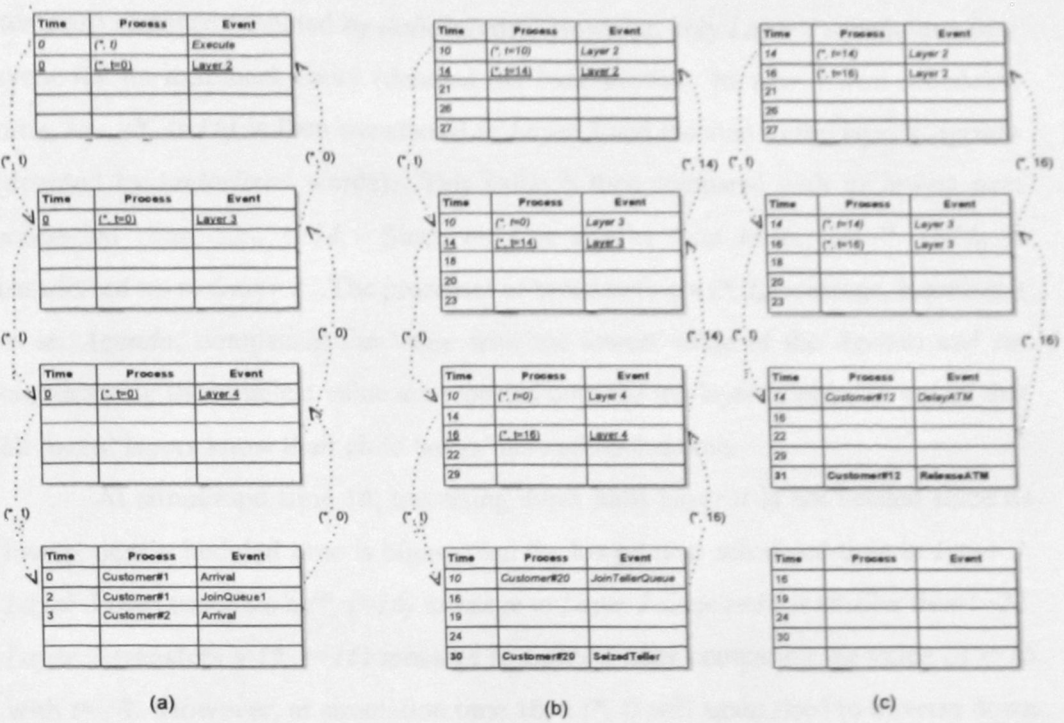


Figure 5.15 Agenda States

At simulation time 0 (i.e., at initial run time), broadcasting a *(*, t)* message down to the lowest layer (i.e., *Layer 4*) is compulsory to find the lowest next scheduled time for the model. This example locates a *Source* component in the *Layer 4*. However, if it were located in other layers, broadcasting the *(*, t)* message down to the lowest layer would ensure the lowest next scheduled time is collected among the *Agendas*.

When the *(*, t)* message reaches the lowest layer, the *(*, t)* is converted to a *(*, t=0)*; we assume that 0 is the first event; i.e., the creation of first entity. The *(*, t=0)* is then transferred up to the top layer since it is the lowest next scheduled time in the whole hierarchy. After this first iteration, each time a *(*, t)* goes down toward its origin layer, all the visited layers' monitors need to execute their *Agendas* by removing their imminent item; i.e., the *(*, t=value)* message. For example, executing the monitors in *Layer 2* and *Layer 3* at simulation time 0 removes the *(*, t=0)* from their *Agendas*. Only *Layer 4* that contains a default entity (which is inserted by the *Source* component) removes the *(*, t=0)* and schedules a new event for the entity.

At simulation time 10, a *(*, t)* message is broadcasted to *Layer 4* from which the *(*, t=10)* has come. During this *(*, t)* broadcasting, all visited *Agendas*' imminent

items are removed (denoted by *italic* words). However, only *Layer 4* schedules a new event for its imminent entity (denoted the *bold* words). Its new lowest scheduled time, i.e., *(*, t=16)* is then transferred to *Layer 3* and inserted to the layer's *Agenda* (denoted by *underlined* words). This value is then compared with its lowest next scheduled time; i.e., *t=14*. Since *t=14* is smaller than *t=16*, the *(*, t=14)* is transferred up to *Layer 2*. The processes of broadcasting a *(*, t_n)* message, inserting it to an *Agenda*, comparing the value with the lowest value of the *Agenda* and re-broadcasting the smallest value are repeated until the top layer in order to ensure that all parent layers know their child layers' next scheduled time.

At simulation time 14, traversing down until *Layer 4* is not needed since its lowest next scheduled time is bigger than the lowest next scheduled time in *Layer 3*. *Layer 3* then transfers a *(*, t=16)* message to *Layer 2* since *t=16* is smaller than *t=22*. *Layer 2* transfers a *(*, t=16)* message to *Layer 1* after comparing the value of *t=16* with *t=18*. However, at simulation time 16, a *(*, t)* will again need to traverse down to the *Layer 4*. These processes will continue until the length of simulation time has been reached.

5.10 Problems and Challenges

The ability to create many visualization instances during runtime can slow model execution and could create awkward model visualization. Model execution is dependent on the number of visualization instances on the stages and more visualization instances will definitely demand more time to render the data on the instances. Awkward model visualization happens when we do not control the depth of the objects on the stage properly. For example, DES components or entity instances that have higher depth than a visualization instance will disturb learners' view of data rendered on the visualization instance whenever it is dragging over them. Thus, we need to specify a range of depth numbers that a certain object type can take whenever it is created.

In order to properly stack objects on the stage, we first gave a lower range of depth numbers for active and passive components, followed by a *Monitor*, a *Utility*, entities and then visualization components. This ensures that all visualization components are always on the top of the stage wherever they are dragged. Entities

should have higher depth compared to simulation components for a reason that they should move over the model structures fabricated by the simulation components.

Based on the *Delegation Event Model*, we can actually permit learners to modify or expand model structures during runtime. This is possible since a simulation component' output port only needs to be fed with the name of its listener in order to transfer entities to the listener. For this, we need to provide a palette that hosts various simulation components (as in our first approach of providing visualization components) where a relevant component can be instantiated with a default ID name by clicking its associated symbol, dragged onto a certain location and linked to its upstream component; e.g., through dragging a point from the instance to the upstream component.

Permitting model configuration during runtime can create interesting activities that engage learners with the model. Observing and analyzing the effect of change of model structures to model behaviour will help learners to understand the model better. However, allowing learners to drag simulation components during runtime will pose a problem; i.e., the animation of entity movement between a component and a dragged component could not be simulated properly. This is true when entities are moving toward the component and at the same time the target component is dragged to other places. As a result, the entities will not properly reach their destination since the distance calculated when they started moving has already changed.

We sometimes need auxiliary messages (in addition to entity messages) for accomplishing relevant tasks in DES; e.g., in activating transporter or handling *reneging* and *jockeying* activities in a queue. Handling reneging and jockeying needs a queue to acknowledge a component that handles these activities, i.e., by sending messages that contain entity names whenever the entities enter the queue. The component needs two main properties: (1) *tolerance time* that employs a list of distributions for representing the time limit that the entity is willing to wait in the queue, and (2) *destination port* for specifying the destination that the entity will go after being retrieved from the queue. A message received from the queue will be delayed based on its tolerance time. When the message has consumed the time, it will search its associated entity in the queue. If its associated entity is still available, the entity will retrieved from the queue and moved to the destination specified in the *destination port*. The message will then be destroyed. If its associated entity is missing (i.e., its associated entity has been removed from the queue), the message will

just be destroyed. We have to insert these auxiliary entities into the model's *Agenda* to tally their execution with the model time. However their existence in the *Agenda* could make the Agenda looks clumsy.

We could use Flash's keyframes to form layers in a hierarchical DES model. Each keyframe handles a sub-model's structures and provides a platform for learners to conduct experiments and customize its visualization. However, Flash treats each keyframe as a totally new program. It only provides a basic transition between keyframes; i.e., moving an execution point from keyframe to keyframe without a support for either accessing objects in or transferring objects to other keyframes. In case of the development of hierarchical DES model, this hinders us from passing entities or other types of messages to other keyframes. Such an approach is totally difference with Microsoft Visual Basic (Wright, 1998) since this language allows the use of *FormName.ObjectName.Property* to access objects that reside in other forms and objects can be passed from form to form freely.

The only way to implement the discussed mechanisms is the use of only one keyframe, but with a number of main movie clips. Each movie clip represents a layer and can contain many other movie clips; i.e., simulation components, visualization components, etc. Since all movie clips now reside in the same keyframe, the simulation components can easily be accessed from other movie clips and the lifecycles of entities and *(*, t)* messages can be maintained. To prevent the clumsiness of many main movie clips on a stage, learners should be allowed to hide or display the main movie clips.

CHAPTER 6

EVALUATION AND ANALYSIS

6.1 Introduction

Learners should acquire knowledge and experiences during their learning. Knowledge can be delivered using various media (e.g., communications, texts, etc.) in classrooms or through online environments. However, experiences can only be gained when learners are exposed to real applications of the knowledge; e.g., through the use of models that implicitly embed the knowledge.

Interactive models can offer learners valuable experiences in two ways: *providing information* explicitly or implicitly during model exploration and *challenging learners' judgment* during model interaction. For example, the explanation of how various variables affect DES systems can offer basic knowledge to learners. However, allowing them to explore and interact with relevant models of the systems will really fill in and clarify their mental models. Thus, the use of various teaching modalities to meet various types of learners' needs is important in learning and teaching settings (Fenrich, 2006; Smith & Renzulli, 1984).

Learning and understanding DES concepts is a challenging task. This is especially true when the availability of teachers in assisting learning is rather limited; e.g., in online environments. There are a lot of static materials that completely explain DES concepts. Although their use in the learning environment has been claimed to have at least equal learning outcomes as interactive materials (e.g., Hegarty, Kriz, & Cate, 2003; Mayer, Hegarty, Mayer, & Campbell, 2005; N. H. Narayanan & Hegarty, 2002; Tversky & Morrison, 2002), they typically fail to attract and engage learners, especially *visual learners* who learn by seeing and visualizing, and *kinaesthetic learners* who learn by doing relevant activities. There are also a lot of attractive DES models. However, they were developed for specific real systems that typically focus on system performance analysis. Since their focuses are more on

final outputs rather than getting insight into model behaviour, interactions with the models are considered as irrelevant aspects.

We believe that queuing models created using our components are attractive, interactive, informative and useful to be used in the learning and teaching environment. The main premise for this claim is that we have designed DES components that are capable of providing models that fulfil characteristics of educational models as suggested in literature (e.g., Bransford, 2000; Lunce, 2004, 2006; Mildrad, 2002). These include *activities* through variable manipulations, *informative and meaningful feedback* through various visualization tools, *attractive animation* of various objects that depicts model behaviour and flexibility in *replicating of real systems*. However, this assumption needs to be assessed through experiments; i.e., by obtaining feedback from a sample of learners about knowledge and insight they gain while experiencing samples of our models. Analyzing the feedback will truly indicate if our tool can construct queuing models that have a positive effect on learning.

We conducted two types of experiments. The first experiment evaluated *learners' perceptions* about the attractiveness and interactivity of samples of our DES models. For this, we designed our own questionnaire based on model characteristics argued important in literature. The second experiment evaluated model *designers' perceptions* about the usefulness, ease of use and enjoyment of the tool and their willingness to use the tool in the future. To measure these factors, we used the Technology Acceptance Model (TAM) and other extension models found in literature. We also assessed the participants' workload while experiencing our tool using NASA Task Load Index (TLX).

6.2 Evaluating Models' Attractiveness and Interactivity

6.2.1. Assessment and Evaluation Methods

We developed our own questionnaire to evaluate the attractiveness and interactivity of models constructed using our component-based tool. The questionnaire was divided into four main sections: general information, general questions, model ratings and additional questions.

The general information section contained two questions: how much computer experience our participants had and how much they used computers as a learning tool. The general questions also consisted of two questions. The first question was based on a five-point Likert-type scale that requested the participant to circle one of available options (i.e., 1 = strongly disagree; 2 = disagree; 3 = neither disagree nor agree; 4 = agree; 5 = strongly agree) that they had good knowledge on simulation. The second question requested them to specify how long they had spent exploring the given models. Thus, during our briefing each participant was reminded to record how long they used the models.

The model ratings are shown in Table 6.1. Items in this section were all based on a five-point Likert-type scale. However, they were invited to write any comment on each of these items. All items were always asked from the positive aspects (i.e., we did not mix positive and negative aspects of items). This makes it easier for them to understand the items and avoids them making any inadvertent mistakes when circling the options from strongly disagree to strongly agree.

The development of the items were based on educational model characteristics that were argued to be important in literature (e.g., Beux & Fieschi, 2007; Gredler, 2003; Jeffries, 2005; Jong, 1991; Joolingen & Jong, 1991a; Swaak & Jong, 2001a). We embedded all these characteristics in our components to produce such types of models. Samples of resulting models were then tested to obtain learners' levels of satisfaction for each criterion so that we can judge the attractiveness, interactivity and usefulness of the models. Note that we did not include item number 12 in Table 6.1 since it contained a list of sub-items that requested the participants to rate if each visualization tool (e.g., graphs, histograms and boxplots) and each facility provided by the models (e.g., ability to pause, resume and adjust animation speed, table of events, etc.) helped them to understand the models better. The item and its sub-items were displayed in Table 6.5.

The additional question section also consisted of two items. The first item asked the participant if they had ever used other animated queuing models. The second item invited the participants to provide additional suggestions on how to make learning through simulation easier.

Table 6.1 Items in Model Rating

1. I am clear about the objectives of the model.
2. The model is useful for information visualization and observing animated objects and events.
3. The model is interactive, inviting input and providing appropriate feedback.
4. The model contains high quality animation which makes learning enjoyable and interesting.
5. The animation helps me to understand scenarios in the model.
6. The various performance visualizations (graphs and other data displays) are meaningful.
7. The model provides a graphical user interface (GUI) which is easy to interact with.
8. I like the design of the GUI.
9. It is good that the visualizations (e.g., graphs, histograms, etc.) are only displayed when requested.
10. The interaction with the model by changing the model's parameters during model execution (e.g., arrival rate, queue rule, server unit) is important in order to understand model behaviour.
11. The change of the representation of animated objects based on their current states is important for me.
13. The model is considerably out of bugs. Please specify if you found any bugs while running the model.
14. Overall, the attractiveness and interactivity of the model is good. Any suggestions to improve the attractiveness and interactivity of the model?
15. I would like to use this kind of model for understanding queuing scenarios.

6.2.2. Experiment Participants

Our objective is to obtain as much as possible of learners' honest feedback about their experiences while using the given models. Thus, we only distributed the models to volunteer participants. Additionally, we did not impose them any time limit and time specification to use the models (i.e., they could explore the models how long the wished at their leisure time). These approaches allowed them to interact with the models and observed the impacts of any changes they had made in a convenient way without any constraints (e.g., unfocused mind, bad mood, etc.). However, since simulations are under constructivist learning, their feedback about the usefulness, attractiveness and interactivity of the models could be influenced by certain factors. These include their types of learners whether they are visual learners, auditory learners, kinaesthetic learners or read-write learners (Aragon, Johnson, & Shaik, 2002; Haapala, 2006), their prior knowledge on a relevant domain (Dochy et al.,

1999; Hailikari et al., 2008; Johnson, Aragon, Shaik, & Palma-Rivas, 2000), etc. Above all, the feedback analyses could give us hints on the participants' acceptance of the models.

We conducted this experiment in a two-week time interval. Participants were approached in the laboratories of the Computer Science and Software Engineering Department, and the laboratories of the Mathematics and Statistics Department (both at the University of Canterbury) for their willingness to participate in the experiment. They were offered an incentive; i.e., two bars of chocolate. A total of 28 participants volunteered to experience our sample models. They were from various year students and programmes; e.g., Computer Sciences, Engineering, Mathematics, Commerce, etc. Six of them were female and the rest were male. We purposely distributed our models to various students so that we had flexibility in analyzing the feedback from various learners about the models' attractiveness and interactivity, irrespective of their knowledge on simulation. This enabled us to analyze the feedback in various angles; e.g., analyzing the data based on overall participants, gender and/or their knowledge levels of simulation.

All of the participants were provided with two models. The first model (Figure 6.1) simulated a simple queuing network. It populated two types of simulation entities using two *Source* components. The first type only required a single server to be processed. The second type needed two servers, the second of which was the same one that processed the first type of entities. The second model (Figure 6.2) just added complexities into the first model. The first type selected an idle server from two parallel servers. After going through one of the parallel servers, they needed to visit another server before leaving the model. The second type selected a server with a shorter queue. After going through this process, only 30% of them directly leave the system. Another 70% went through the servers that processed the first type of entities. However, they did not need to go through another server as for the first type of entities; instead they directly left the model. See Appendix C.

The purposes of the experiment and the description with a snapshot of each model were provided on an information sheet and attached to the questionnaire. Additionally, we demonstrated the models to each participant and explained what they were requested to do during and after the experiment (e.g., clicking components, changing their variables, instantiating visualization tools, changing animation speed, etc.) so that they had some strategies in their exploration. This was important since

the models were open-ended simulation models that needed the participants to at least be equipped with basic mental models before they were left free to explore the models themselves. They were also briefly introduced to all items in the questionnaire in order to make sure that they understood the items and answered them appropriately. Any relevant questions regarding the models and the questionnaire were then welcomed and answered.

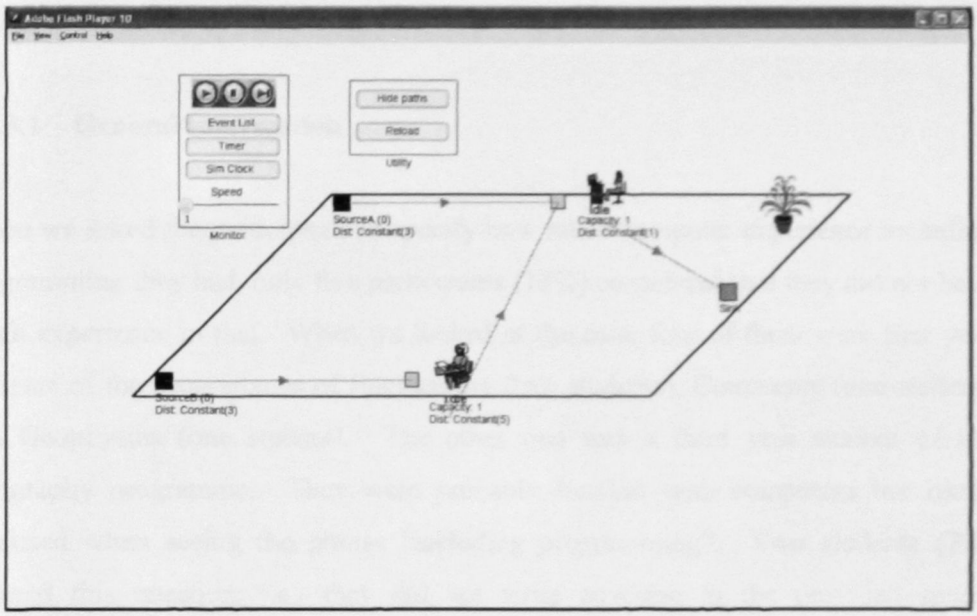


Figure 6.1 Simple Queuing Networks

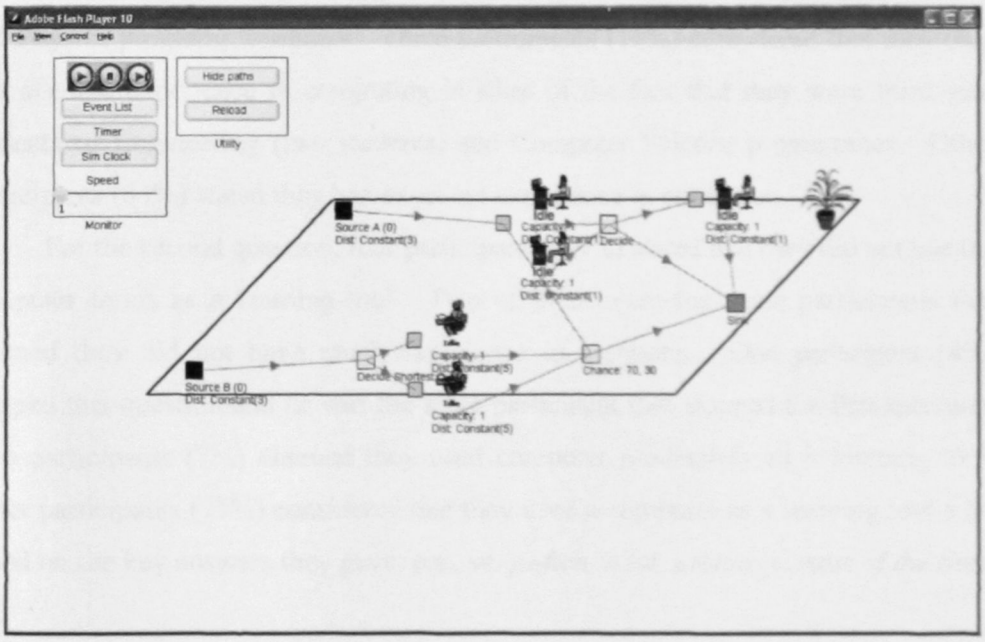


Figure 6.2 More Complicated Queuing Networks

The participants were encouraged to experience with both of the models. They were then left to use the models as long as they wished either in the laboratories or at their homes. By leaving the models to be experienced at their leisure they had and no time limits imposed, we hoped that we would get as honest feedback as possible.

6.2.3. Data Analysis and Results

6.2.3.1 General Information

When we asked the participants to specify how much computer experience including programming they had, only five participants (18%) considered that they did not have much experience in that. When we looked at the data, four of them were first year students of the programmes of Engineering (two students), Commerce (one student) and Geophysics (one student). The other one was a third year student of the Geography programme. They were probably familiar with computers but likely confused when seeing the phrase "including programming". Two students (7%) skipped this question; i.e., they did not write anything in the provided space. However, we believed that both of them had quite experience in programming since they were a fifth year Engineering programme student and a third year Mathematics and Physics programme student. Three participants (11%) considered that they only had average experience in computing in spite of the fact that they were third year students of Engineering (two students) and Computer Science programmes. Other participants (64%) stated they had excellent experience in computer.

For the second question, four participants (14%) stated that they did not use the computer much as a learning tool. Two of them were the same participants that claimed they did not have much experience in computer. One participant (4%) skipped this question and he was the same participant that skipped the first question. Two participants (7%) claimed they used computer moderately as a learning tool. Other participants (75%) considered that they used a computer as a learning tool a lot based on the key answers they gave; e.g., *very often*, *a lot*, *everyday*, *most of the time*, etc.

6.2.3.2 General Questions

Simulation is a learning environment where its contents are not explicitly exposed to learners. Its usefulness in providing the opportunity to learn in a more realistic context heavily depends on students' prior knowledge. There are two types of knowledge that learners should have: *specific conceptual knowledge*; i.e., the domain-specific knowledge about concepts and facts that a model represents, and *general knowledge*; i.e., quantitative and qualitative aspects to read information and draw conclusions from the model's outputs. The importance of both types of knowledge in structuring and accommodating learning through models has been argued in much literature (e.g., Dochy et al., 1999; Hailikari et al., 2008).

Operating a simulation model without the knowledge may create three distinct problems. First, learners tend to conduct inefficient experiments, thus any interactions with the model seems not to be important. Second, learners may have trouble in interpreting information, thus animations and data visualizations seem to give insignificant impacts and eventually demotivate them to learn. Third, learners may not be able to regulate their learning processes, thus the model seems not to be useful. Therefore, collecting participants' prior knowledge to properly judge their feedback about the usefulness of our models and their relevant features in ensuring the participants' learning is important.

Based on the participants' responses, only six participants (21%) were confident (agreed/strongly agreed) that they had good knowledge on simulation. Nine participants (32%) considered that they did not have good knowledge on simulation based on their choices of strongly disagree/disagree options. The other thirteen participants (46%) stated that they were undecided about their knowledge on simulation. Figure 6.3 shows the frequencies of the participants' scores for the first general question.

Table 6.2 shows the summary reports of estimated time spent on the models by all participants grouped by their knowledge levels on simulation. The average time spent by all of the participants was 17.61 minutes. The minimum and the maximum time spent were 3 minutes and 60 minutes respectively. Both the minimum and the maximum values were from the participants that were undecided about their knowledge on simulation.

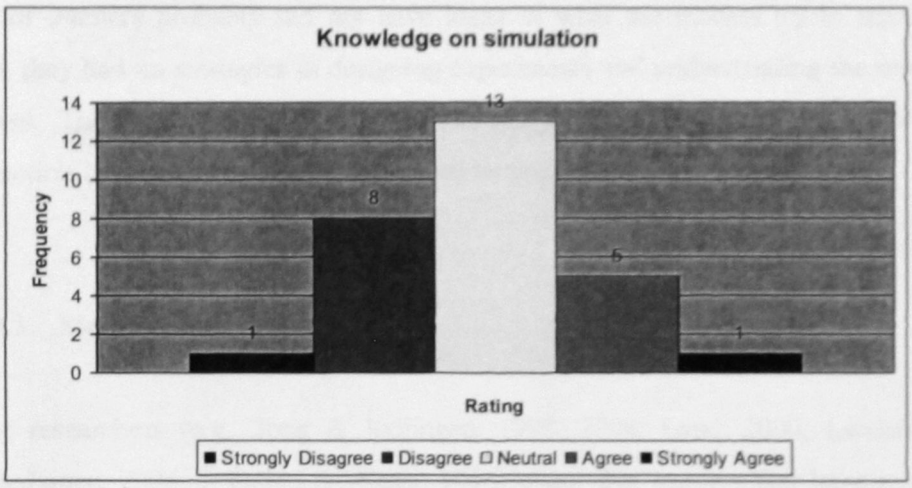


Figure 6.3 Participants' Feedback on Simulation Knowledge

Table 6.2 Time Spent (in minutes) for Each Level of Knowledge on Simulation

Score	N	Minimum	Maximum	Mean	Std. Deviation
1	1	10	10	10	-
2	8	5	30	13.75	7.44
3	13	3	60	19.08	19.26
4	5	10	30	23	9.75
5	1	10	10	10	-

It is interesting to observe that the participants who agreed that they had good knowledge on simulation were in fact the group that used the models for the longest time in average (i.e., 23 minutes), followed by the group of participants that neither disagreed nor agreed that they had good knowledge on simulation (i.e., 19.08 minutes). This perhaps signals that the use of simulation models in learning settings is effective for learners for whom their knowledge levels on simulation are between moderate and good. One possible reason for this is that learners in this group more often have hypotheses in mind to be tested during their exploration. These induce them to engage with the models through conducting and understanding the models' relevant outputs.

If we look at Table 6.2, the use of simulation models could probably fail to engage the extreme point participants; i.e., the participants that had little knowledge on the concepts that the models represented and the participants that had already had

concrete mental models about the concepts. One reason for the former is that this type of learners probably did not have ideas of what the models try to represent. Thus, they had no strategies in designing experiments and understanding the models' outputs. The reason for the latter is that this type of learners probably felt bored with the models because their outputs could well be predicted for each experiment.

6.2.3.3 Model Rating

Some researchers (e.g., Jong & Joolingen, 1998, 2008; Land, 2000; Landriscina, 2009; Lunce, 2006; J. Quinn & Alessi, 1994) claim that learners that have relevant mental models or been equipped with some basic knowledge can effectively experience and evaluate open-ended simulation models. Based on this argument, we separated our analyses based on the participants' knowledge on simulation. Table 6.3 reports the experienced participants' (i.e., who had good simulation knowledge) feedback about the models. Table 6.4 and Table 6.5 meanwhile report the inexperienced participants' feedback about the models; i.e., who were undecided and who did not consider that they have good simulation knowledge respectively. By separating the results, we can effectively evaluate and judge the usefulness of our models in offering the opportunity to learn DES concepts and the significance of their features in ensuring the participants' learning.

Question 1 asked the participants if they were clear about the objectives of the models; i.e., what situations the models represented and what they were expected to gain while exploring the models. Interestingly, all the six experienced participants were clear about the objectives of the models. This indicated that they had a clear picture about the principles of the models. Of the nine participants who claimed that they did not have knowledge on simulation, only one participant (11%) was unclear about the objectives of the models. There were two participants (22%) undecided while the remaining six participants (67%) stated that they understood the model objectives. Of the group that were undecided about their knowledge on simulation, six participants (46%) confirmed that they were clear about the objectives of the models. Only two participants disagreed with this statement. In general, most of the inexperienced participants (55%) understood the purposes of the models. We believed that our approach of providing a description sheet of the models,

demonstrating the models and handling a *question and answer* session with the participants before they started their explorations gave some mental images for most of the participants in these two groups.

Table 6.3 Good Simulation Knowledge Participants' Feedback about the Models

Item	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Clear objectives	0 (0%)	0 (0%)	0 (0%)	4 (67%)	2 (33%)	4	4.33	0.21
Model useful	0 (0%)	0 (0%)	0 (0%)	3 (50%)	3 (50%)	4, 5	4.50	0.22
Model interactive	0 (0%)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	5	4.33	0.33
Quality animation	0 (0%)	0 (0%)	1 (17%)	4 (67%)	1 (17%)	4	4.00	0.26
Animation helpful	0 (0%)	0 (0%)	0 (0%)	3 (50%)	3 (50%)	4	4.50	0.22
Visualization meaningful	0 (0%)	0 (0%)	0 (0%)	4 (67%)	2 (33%)	4	4.33	0.21
GUI interactive	0 (0%)	0 (0%)	1 (17%)	4 (67%)	1 (17%)	4	4.00	0.26
GUI acceptable	0 (0%)	0 (0%)	1 (17%)	5 (83%)	0 (0%)	4	3.83	0.17
Pop-up visualization	0 (0%)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	5	4.33	0.33
Interaction helpful	0 (0%)	0 (0%)	1 (17%)	3 (50%)	2 (33%)	4	4.17	0.31
Animation important	0 (0%)	1 (17%)	0 (0%)	3 (50%)	2 (33%)	4	4.00	0.45
Model out of bugs	0 (0%)	0 (0%)	3 (50%)	2 (33%)	1 (17%)	3	3.67	0.33
Model good	0 (0%)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	5	4.33	0.33
Model preference	0 (0%)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	5	4.33	0.33

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Table 6.4 No Simulation Knowledge Participants' Feedback about the Models

Item	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Clear objectives	0 (0%)	1 (11%)	2 (22%)	6 (67%)	0 (0%)	4	3.56	0.24
Model useful	0 (0%)	1 (11%)	0 (0%)	7 (78%)	1 (11%)	4	3.89	0.26
Model interactive	0 (0%)	1 (11%)	1 (11%)	6 (67%)	1 (11%)	4	3.77	0.28
Quality animation	0 (0%)	0 (0%)	3 (33%)	4 (44%)	2 (22%)	4	3.89	0.26
Animation helpful	0 (0%)	0 (0%)	3 (33%)	5 (56%)	1 (11%)	4	3.78	0.22
Visualization meaningful	0 (0%)	2 (22%)	1 (11%)	6 (67%)	0 (0%)	4	3.44	0.29
GUI interactive	0 (0%)	1 (11%)	4 (44%)	3 (33%)	1 (11%)	3	3.44	0.29

GUI acceptable	0 (0%)	0 (0%)	5 (56%)	3 (33%)	1 (11%)	3	3.56	0.24
Pop-up visualization	0 (0%)	1 (11%)	2 (22%)	5 (56%)	1 (11%)	4	3.67	0.29
Interaction helpful	0 (0%)	0 (0%)	1 (11%)	6 (67%)	2 (22%)	4	4.11	0.20
Animation important	0 (0%)	2 (22%)	2 (22%)	5 (56%)	0 (0%)	4	3.33	0.29
Model out of bugs	0 (0%)	0 (0%)	5 (56%)	1 (11%)	3 (33%)	3	3.78	0.32
Model good	0 (0%)	0 (0.00%)	2 (22%)	6 (67%)	1 (11%)	4	3.89	0.20
Model preference	0 (0%)	1 (11%)	2 (22%)	3 (33%)	3 (33%)	4	3.89	0.35

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Table 6.5 Undecided Simulation Knowledge Participants' Feedback about the Models

Item	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Clear objectives	0 (0%)	2 (15%)	5 (38%)	2 (15%)	4 (31%)	3	3.62	0.31
Model useful	0 (0%)	0 (0%)	2 (15%)	10 (77%)	1 (8%)	4	3.92	0.14
Model interactive	0 (0%)	1 (8%)	4 (31%)	5 (38%)	3 (23%)	4	3.76	0.26
Quality animation	0 (0%)	4 (31%)	4 (31%)	4 (31%)	1 (8%)	2, 3, 4	3.15	0.27
Animation helpful	0 (0%)	0 (0%)	2 (15%)	7 (54%)	4 (31%)	4	4.15	0.19
Visualization meaningful	0 (0%)	1 (8%)	5 (38%)	5 (38%)	2 (15%)	3, 4	3.62	0.24
GUI interactive	0 (0%)	3 (23%)	5 (38%)	3 (23%)	2 (15%)	3	3.31	0.29
GUI acceptable	1 (8%)	2 (15%)	4 (31%)	5 (38%)	1 (8%)	4	3.23	0.30
Pop-up visualization	0 (0%)	0 (0%)	1 (8%)	6 (46%)	6 (46%)	4, 5	4.38	0.18
Interaction helpful	0 (0%)	0 (0%)	3 (23%)	6 (46%)	4 (31%)	4	4.08	0.21
Animation important	0 (0%)	2 (15%)	3 (23%)	7 (54%)	1 (8%)	4	3.54	0.24
Model out of bugs	0 (0%)	1 (8%)	4 (31%)	5 (38%)	3 (23%)	4	3.77	0.26
Model good	0 (0%)	2 (15%)	3 (23%)	6 (46%)	2 (15%)	4	3.62	0.27
Model preference	0 (0%)	1 (8%)	2 (15%)	9 (69%)	1 (8%)	4	3.77	0.20

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Much literature (e.g., Falvo, 2008; Hegarty, 2004; Hegarty et al., 2003; Lowe, 2004) stresses the usefulness of embedding animations and data visualizations in educational models. Animations motivate learners to learn and help them get insight

into complicated phenomena and understand the relationships between various model variables. The effect of these variables to model behaviour is then made visible through various data visualizations. However, the usefulness of animations and data visualizations is much influenced by whether or not a learner has been equipped with basic domain specific knowledge for understanding model outputs, generic knowledge of quantitative and qualitative methods for interpreting the outputs and skills for performing further experiments.

Question 2 tested if our models were useful for information visualizations and observing animated objects and events in order to understand the models' states and behaviour. All of the six experienced participants believed that the models were useful for these. This reflected that our DES components could build models with good animations and data visualizations. Data also revealed that eight of the participants (89%) who did not have good knowledge on simulation and eleven of the participants (85%) who were undecided about their simulation knowledge considered that our models provided useful animations and information visualizations. Of these inexperienced participants, only one participant disagreed with the statement. The high percentage of agree/strongly agree opted by the participants in this group showed that information visualizations and animations of objects and events in our models helped them understand DES concepts better.

Interaction plays an important role in any learning processes (Arbaugh & Benbunan-Fich, 2007; Su et al., 2005; Woo & Reeves, 2007). In the traditional classroom environment, interactions between learners and their teachers can stimulate their knowledge acquisition and clarify their judgment. In case of virtual classrooms and online learning environments that use models as mediums of instructions, model interactivity can replace the teachers' role. Although this feature does not guarantee learning through models (Davies, 2002; Pilkington & Parker-Jones, 1996), its significance in motivating and engaging learning has been corroborated in many studies (e.g., Beux & Fieschi, 2007; Bransford, 2000; Mildrad, 2002; Schank, Berman, & Macpherson, 1999). Question 3 tested if our models were interactive, inviting input and providing appropriate feedback.

Based on the data, five experienced participants (83%) agreed/strongly agreed that our models were interactive and provided appropriate feedback. The other one participant circled an undecided option. This indicated that DES models built using our components provided an interactive platform for stimulating active explorations

and showing cause and effect of the participants' relevant actions. Of the inexperienced participants, only two participants (9%) disagreed with the statement; one was from the participants that disagreed that she had good knowledge on simulation while the other one was the participant that was undecided about his knowledge on simulation. A majority of participants that did not have knowledge on simulation (i.e., 78%) and were undecided about their knowledge on simulation (i.e., 62%) agreed that the models were interactive. Once again, the feedback reflected that our models were interactive and informative to be used as DES learning tools even though they were used by the participants that did not have adequate prior knowledge on DES.

Flash has been claimed to produce high quality animated applications (Castillo et al., 2004; Mohler, 2006; Shupe & Hoekman, 2006). This was a reason why we used Flash to build DES models and animate their behaviour. Question 4 tried to obtain feedback from the participants about the animation quality of our models. Five experienced participants (83%) considered that the models contained high quality animations. Of the inexperienced participants, only four participants (18%) disagreed with the statement and they were the participants that were undecided about their simulation knowledge.

Table 6.6 shows in details the feedback of the participants that claimed they often used computer as a learning tool (21 participants) about the animation quality of our models. Eleven participants (52%) from this group considered that the models contained high quality animations. Only three participants (14%) disagreed with the statement. The majority of agreed/strongly agreed participants indicated that our components produced high quality animated models that could effectively represent the DES concepts which were difficult to be explained in static materials. The animations offered exciting learning materials that motivated their learning and attracted them to engage with the models.

Table 6.6 Feedback on the Quality of Animation from the Participants Who Always Used Computer as a Learning Tool

Scale	Frequency	Percent	Valid Percent	Cumulative Percent
2	3	14.3	14.3	14.3
3	7	33.3	33.3	47.6
4	8	38.1	38.1	85.7
5	3	14.3	14.3	100.0
Total	21	100.0	100.0	

Model presentation is important to attract and engage learners (Djajadiningrat, Matthews, & Stienstra, 2007; Parrish, 2009). The use of meaningful animations for showing model behaviour can offer many benefits. These include facilitating learners' understanding about dynamic processes in a model, making the learning experience enjoyable and enriching. Some studies have also shown that learning through meaningful animations typically motivates learners to learn and induce them to retain information longer (Teoh & Neo, 2007; Vogel-Walcutt, Gebrim, & Nicholson, 2010). Question 5 tested if our embedded animations helped them to understand scenarios in the models. Interestingly, all the experienced participants agreed/strongly agreed with this statement. The feedback reflected that our approach of demonstrating the behavior of the models through meaningful animations (e.g., showing a sequence of events, animating the movement of entities and their current states, changing the pictures of a server based on its status, etc.) was very useful for understanding the models. Data also revealed that six of the participants (67%) who did not have knowledge on simulation and eleven of the participants (85%) who were undecided about their knowledge on simulation agreed/strongly agreed with the statement. This suggested that we successfully integrated animations in our DES models and the animations helped this inexperienced group understand scenarios in the models.

When asked if various performance visualizations were meaningful for learning (Question 6), all the experienced participants gave positive feedback on the item. This showed that graphs and other data displays used to report the detailed performance of the models over simulation time were meaningful and should be used to complement animations. This is expected since this group of learners knows the importance of the visualization tools in measuring the performance of the models. However, three of the inexperienced participants (14%) disagreed and six of them (27%) were undecided about the meaningful of the various performance visualization tools. This probably signaled that the visualization tools may not so useful unless learners would like to understand in details the current performances of the models.

GUIs play important roles in data-driven simulations; i.e., to capture learners' inputs and send them to particular model processes. We partitioned the processes to relevant components, each of which has its own GUI that can be accessed by clicking on it. The GUIs have two functions: (1) displaying all editable variables and their current values, and (2) instantiating data visualization tools that graphically chart the

component behavior in real time. We expected this approach enabled learners to easily interact with the models (Question 7). Data analysis showed that five of the experienced participants (83%) agreed/strongly agreed with us. Of the inexperienced participants, only four (18%) disagreed that the GUIs provided by our tools were easy to interact with. The results might indicate that the use of a *mouse clicking* approach to access components offered an easy platform for learners to explore and experiment with the models. However, a better approach to access the GUIs should be investigated since about half of the inexperienced participants were still undecided if the GUIs were easy to access.

When asked if they liked the design of the GUIs (Question 8), five of the experienced participants (83%) agreed with the statement. This might reflect that our approach of providing simple interfaces using text boxes, command buttons, combo boxes, etc. and presenting simulation results in various windows that can be dragged to any locations was effective. However, three inexperienced participants (14%) did not like the design of the GUIs. One of them was the same participant that disagreed the GUIs were easy to interact with. The other two participants were from the participants that could not decide if the GUIs were easy to interact with. Interestingly, there were no participants that agreed/strongly agreed that the GUIs were easy to interact with did not like the design of the GUIs.

There has been a substantial amount of evidence that proves the use of multiple representations through different choices of data presentations and different forms of feedback can significantly enhance learning in complex domains (e.g., see Ainsworth, 1999; Ainsworth, Bibby, & Wood, 2002; Bodemer & Faust, 2006; Goldman, 2003; Kozma, 2003; Schnotz & Bannert, 2003; Seufert, 2003). Unfortunately, this desirable feature has not been integrated in DES models. Our DES models allow visualization customizations; i.e., learners can dynamically create a number of visualization instances from many available types of visualization tools (e.g., graphs, tables, clocks, etc.) during a simulation run. Thus, our models can be represented by many interfaces, with each interface containing many representations that show various angles of model information and variable relationships. For examples, texts are used to represent certain contexts, graphs (or other visualization tools) or tables of numeric values are used to represent quantitative aspects of the models and animations are used to represent qualitative information of their inner processes. Data analysis of Question 9 showed that five experienced participants

(83%) considered that the approach of displaying visualizations only when requested was a good approach. Of the inexperienced participants, only one participant disagreed with this approach. A majority of them showed their strong support for the approach. The feedback reflected that our approach of allowing learners to customize their own visualizations was deemed as a good idea since they could control the display of model information based on their ability to understand the models' behaviour.

As mentioned earlier, interactions during model execution are important to understand model behaviour. However, most DES models provide no support for model variable alterations during runtime. This is totally different with our DES models that allow learners to interact with DES variables (e.g., by changing arrival rates, queue rules, server units, etc.) on the fly and observe the effect of those variables to model behaviour. Question 10 tests if this approach is important in learning. Five experienced participants (83%) stated that this feature helped their learning. Of the inexperienced participants, there was no one who was negative about the importance of this approach (although there were four participants (18%) could not decide). This proved that providing an interaction platform for learners to clarify their ideas was a desirable feature for learning through models.

The change of animated object representations explicitly shows the change of model states. We suspected that these tiny changes may not help learners to understand model behaviour so much. However, analyses of Question 11 showed that four experienced participants (67%) agreed/strongly agreed that such changes were important for them to understand model behaviour. Of the inexperienced participants, there were thirteen participants (59%) agreed/strongly agreed while only four participants (18%) disagreed with the statement. This indicated that animations of objects based on their states might assist learning and offered the advantage of delivering better representations of relevant concepts. Thus, animations should be used to explicitly explain dynamic and complicated processes such as DES and system dynamic.

While visualization tools are important to graphically chart the pattern of numerical data, other relevant tools can also offer benefits in easing learning. For example, we provided a slider to allow learners to control animation speed based on their abilities in extracting information from the models (i.e., time scale of events), a table of events to show a list of types of the previous, current and next events with

their occurrence time in relation to model variables, tables of statistical information to report the current statistics of each component, a description table of each entity to display a list of its performed activities in the models, and a facility button to hide and display paths of entity movement. This feature enables them to clearly view the lifecycles of various entities especially for more complex structure models.

The usefulness of these tools in helping learners to understand queuing models was investigated in Question 12. The question was divided into sub-questions, each of which requested the participants to rank the tool's usefulness in model exploration. The sub-questions and their associated tools are shown in Table 6.7. Table 6.8, Table 6.9 and Table 6.10 meanwhile show the descriptive analysis of the participants' feedback about the tools based on their knowledge on simulation.

Table 6.7 Sub-questions of "These tools help to understand the model better (Please write if you have any comments)"

Sub-question	Tool
12.1	Graphs
12.2	Histograms
12.3	BoxPlots
12.4	Ability to pause, resume and adjust animation speed
12.5	Table of events (previous, current and future)
12.6	Table of component's statistical information (e.g., queue, server, etc.)
12.7	Entities' information window showing activities they have performed in the model
12.8	Ability to hide and show the path of entities

Table 6.8 Good Simulation Knowledge Participants' Feedback about the Model Tools

Tool	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Graphs	0 (0%)	0 (0%)	0 (0%)	4 (67%)	2 (33%)	4	4.33	0.21
Histograms	0 (0%)	0 (0%)	1 (17%)	3 (50%)	2 (33%)	4	4.16	0.31
Boxplots	0 (0%)	0 (0%)	1 (17%)	4 (67%)	1 (17%)	4	4.00	0.26
Animation control	0 (0%)	0 (0%)	1 (17%)	1 (17%)	4 (67%)	5	4.50	0.34
Event table	0 (0%)	1 (17%)	1 (17%)	4 (67%)	0 (0%)	4	3.50	0.34
Statistical tables	0 (0%)	0 (0%)	1 (17%)	2 (33%)	3 (50%)	5	4.33	0.33
Information windows	0 (0%)	0 (0%)	2 (33%)	3 (50%)	1 (17%)	4	3.83	0.31
Path visibility	0 (0%)	0 (0%)	4 (67%)	1 (17%)	1 (17%)	3	3.50	0.34

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Table 6.9 No Simulation Knowledge Participants' Feedback about the Model Tools

Tool	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Graphs	1 (11%)	0 (0%)	1 (11%)	6 (67%)	1 (11%)	4	3.66	0.37
Histograms	1 (11%)	0 (0%)	4 (44%)	3 (33%)	1 (11%)	3	3.33	0.37
Boxplots	1 (11%)	1 (11%)	4 (44%)	2 (22%)	1 (11%)	3	3.11	0.39
Animation control	0 (0%)	0 (0%)	0 (0%)	4 (44%)	5 (56%)	5	4.56	0.18
Event table	1 (11%)	2 (22%)	1 (11%)	3 (33%)	2 (22%)	4	3.33	0.47
Statistical tables	0 (0%)	0 (0%)	3 (33%)	4 (44%)	2 (22%)	4	3.89	0.26
Information windows	0 (0%)	1 (11%)	5 (56%)	2 (22%)	1 (11%)	3	3.33	0.29
Path visibility	0 (0%)	4 (44%)	2 (22%)	1 (11%)	2 (22%)	2	3.11	0.42

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Table 6.10 Undecided Simulation Knowledge Participants' Feedback about the Model Tools

Tool	SD	D	NDA	A	SA	Mode	Mean	Std. Deviation
Graphs	0 (0%)	2 (15%)	2 (15%)	9 (69%)	0 (0%)	4	4.57	0.97
Histograms	0 (0%)	1 (8%)	5 (38%)	6 (46%)	1 (8%)	4	3.54	0.22
Boxplots	0 (0%)	3 (23%)	4 (31%)	6 (46%)	0 (0%)	4	3.23	0.23
Animation control	1 (8%)	0 (0%)	1 (8%)	7 (54%)	4 (31%)	4	4.00	0.30
Event table	0 (0%)	4 (31%)	2 (15%)	5 (38%)	2 (0%)	4	3.38	0.31
Statistical tables	0 (0%)	1 (8%)	3 (23%)	6 (46%)	3 (23%)	4	3.84	0.25
Information windows	1 (8%)	1 (8%)	2 (15%)	8 (62%)	1 (8%)	4	3.54	0.30
Path visibility	1 (8%)	2 (15%)	1 (8%)	5 (38%)	4 (31%)	4	3.69	0.36

SD=Strongly Disagree, D=Disagree, NDA=Neither Disagree nor Agree, A=Agree, SA=Strongly Agree

Based on Table 6.8, graphs were rated as the most important visualization tool by the experienced participants (i.e., all of them agreed/strongly agreed that graphs helped them understand the models better), followed by an animation control (five participants with four of them strongly agreed), statistical tables (five participants with three of them strongly agreed), histograms (five participants with two of them strongly agreed), boxplots (five participants with one of them strongly agreed), event tables (four participants) and lastly the path visibility facility (two participants). This reflected that graphs plotting relevant variables (e.g., number of entities in a queue,

number of units of a resource used, etc.) over simulation time and an animation control slider providing a feature for pausing, resuming and adjusting animation speed based on the participants' abilities to retrieve information from simulation were the two most desirable visualization tools to get insight to the models' behaviour. The two visualization tools that received minimum scores were the path visibility facility and event tables. The probable reason why the facility to hide and display received the lowest score was because the models' structures were not so complicated. This tool would be useful if the models' structures were complicated; i.e., they contained many types of entities, each of which has its own paths. The participants that disagreed with the usefulness of the table of events in helping them understand the models claimed that the table was not very human readable. The table was actually used by the models to update their behaviour and it could be used by interested participants to trace how the models' behaviour and their animations have been and will be simulated over time.

For the inexperienced participants, the animation slider was rated as the most important tool (i.e., twenty participants with nine of them strongly agreed), followed by graphs (sixteen participants), statistical tables (fifteen participants), path visibility (twelve participants with five of them strongly agreed), event tables (twelve participants with four of them strongly agreed), information windows (twelve participants with two of them strongly agreed), histograms (eleven participants) and boxplots (nine participants). One inexperienced participant that felt the entity's information window was not an essential feature complained that the windows were hard to locate while in use. This is probably true since the images of the entities in our models are quite small.

When asked if the models were free of bugs (Question 13), fifteen of the participants (three experienced participants and twelve inexperienced participants) agreed/strongly agreed with the statement. Twelve of the participants (three experienced participants and nine inexperienced participants) could not decide while one participant disagreed. Five participants reported two bugs during their exploration. However, two of them still agreed that the models were considerably free of bugs, while the other three participants opted to choose neither disagree nor agree options. These two bugs were: (1) arrows depicting paths of entity movement disappeared after certain simulation time, and (2) certain components sometimes could not be clicked to access their GUIs. One participant complained that the

description texts of some components in the second model were located under other components and this hindered him to properly read the texts. Overall, only one participant disagreed that the models were free of bugs. This reflected that our approach of structuring all classes for the DES components prior to writing their code led to relatively few syntax and logical errors.

We scrutinized our code to find the reasons for these bugs. The first bug happened because we did not properly control the depths of arrow clips connecting the components. After a particular number of depths, the arrows would disappear whenever their depths were replaced by the depths of newly generated entities. We corrected this bug. We however could not find the reasons for the second bug. For the complaint that there were some texts under certain components, we actually overlooked the arrangement of the components in the second model. When simulation structures are getting complex, all simulation components have to be compacted in a limited stage to give learners enough spaces to customize the models' visualization during run time. As a result, texts for some components may be located under some other components.

When asked to rate the overall attractiveness and interactivity of the models (Question 14), five experienced participants (83%) chose agree/strongly agree options. Of the inexperienced participants, fifteen of them (68%) agreed with the statement. This showed that a good balance between quantitative analyses through data visualizations and qualitative aspects through animations, clear presentation and attractive interfaces could improve learners' understanding on DES concepts. There were two participants who disagreed that our models were attractive and interactive, and they were actually the same participants that disagreed that the models contained high quality animations.

Question 15 asked if the participants would like to use these types of models for understanding queuing networks. Five experienced participants and sixteen inexperienced participants (six participants were from the participants that did not consider to have good knowledge on simulation and ten participants were from the participants that were undecided about their knowledge on simulation) would like to do so. A majority of the participants that agreed with the attractiveness and interactivity of the models reflected that our models could be used as self-study or supplementary materials to learn DES concepts. However, there were two inexperienced participants who disagreed that they would use the models. One of

them was the same participant that felt the overall attractiveness and interactivity of the models was not good. This might signal that without basic knowledge, attractive and interactive models would not help and improve students' learning through models.

For the first additional question that asked the participants if they had ever used any other animated simulation models for queuing scenarios, only one participant claimed that she used to use animated simulation models. She stated that the other models that she had used had better graphics but with no exploration capabilities. Five participants (two experienced participants and three inexperienced participants) explicitly noted that our models helped them to understand DES concepts. One participant said that it was so interesting to see the mechanism of queuing networks that were difficult to illustrate using traditional paper-based or static materials.

We invited the participants to suggest how to make simulation learning easier. Some participants responded to this request. Their suggestions included (1) showing the functionality of each component used in the models (e.g., in the form of tool tip texts, pop-up windows, etc.) whenever learners selected the component, (2) providing editable models so that their structures can be changed or modified (e.g., learners can arrange the flow of entities during runtime), (3) providing tutorials or helping menus to assist them whenever they were stuck in their learning processes, (4) providing 3D versions of the models to make them more attractive, and (5) displaying overall results whenever simulation had finished. Some participants noted that our approach of allowing them to create multiple visualizations themselves (i.e., controlling the amount of visualization tools to be displayed and dragging them to wherever locations on the model stage) was really a good approach in helping them to understand model behaviour.

The first suggestion is easy to implement. In fact, we used this approach for showing an entity's activities. Since we implemented a *click* event in a component's code to access its GUI, a *mouse-over* event (that activates a new movie clip and holds a description of its functionality) and a *mouse-out* event (that removes the movie clip whenever a mouse pointer is not on the component region) can be used. The second suggestion can also be accomplished since we implemented the *Delegate Event Model* pattern that uses ports to link components. For this, we need to reveal all components' names and provide fields in their GUIs to accept their downstream

component names during runtime. However, this will make the models look clumsy with component names and prone to logical errors if the output ports are not specified correctly by learners.

We agree that providing a textual tutorial, integrating other multimedia resources or supplying a list of instructions (i.e.; some suggested hands-on experiments) is important to assist learning through models. Examples of hands-on experiments include *investigation experiments* that request learners to investigate the effects of various variables to model behaviour and *optimization experiments* that request learners to identify and vary simulation variable values so that specified model constraints are not broken.

The suggestion of using 3D models to make learning through simulation models easier is not always true. Such models could attract and engage learners since they are close to their actual systems. However, their use in education has been claimed to only benefit some learners while other learners may suffer additional cognitive workloads (Huk, 2006; Korakakis, Pavlatou, Palyvos, & Spyrellis, 2009). To be effective, a simulation model should offer an interactive platform for *hypotheses testing* (i.e., an experimentation platform for clarifying learners' ideas) instead of graphic sophistication that is fun to look (Prensky, 2001).

We have to stress the danger of misinterpretation of DES results by learners manipulating model parameters interactively during simulation run. The animations and visualizations of our models only reflect the impacts of the parameter settings to their current behaviour. They are not supposed to be used as an analysis tool for measuring model performances which strictly requires unchanged parameter values until the end of simulation. The statistical analyses in our models is to help learners understand how a relevant parameter (e.g., time between arrival, route time, queue rule, process time, etc.) affects the models' current states and performance. Some of the analyses can be viewed through animations and visualizations. For example, learners can observe the animations of the current number of entities in a queue and visualize the current utilization of a server. Other analyses are to give the detail of the models' current performance measures over simulation time, and these are typically reported using tables; e.g., throughput, waiting time in a queue (average, minimum and maximum), length in a queue (average, minimum and maximum), time spent in the system (average, minimum and maximum), resource utilization, etc.

6.3 Evaluating the Tool's Ease of Use, Usefulness and Enjoyment

6.3.1 Assessment and Evaluation Methods

Human behaviour has long been claimed as an important element that determines the acceptance of a technological innovation (Greenbaum & Kyng, 1991; Isomaki, Pekkola, & Bannon, 2011). In order to empirically assess model builders' perception towards our component-based tool, we have conducted an experiment by adapting the Technology Acceptance Model (TAM) developed by Davis (1989). Results of this can signal the acceptance of our tool and can be used to improve it in the future.

TAM consists of a list of items (variables) discriminated under two cognitive responses (factors); i.e., *perceived usefulness* and *perceived ease of use*. Perceived usefulness relates to significant functions that the innovation provides while perceived ease of use generally relates to interfaces and attractiveness of the innovation. These responses were originally proposed by the *Theory of Reasoned Action* (Ajzen & Fishbein, 1980; Fishbein & Ajzen, 1975) and significantly determine users' acceptance (i.e., their attitudes and behaviour) of an innovation.

Variables for each factor in TAM were derived from previous empirical studies on the self-efficiency theory (Banduras, 1977), the cost-benefit paradigm (Payne, 1982) and the adoption of innovations (Tornatzky & Klein, 1982). Each factor initially consisted of 14 candidate variables. However, after being tested for reliability and content validity, the variables were then cut out to only six variables (see Table 6.11) that are adequate for testing perceived usefulness and perceived ease of use of an innovation.

TAM has been tested as a valid and reliable model for measuring users' acceptance of an innovation (e.g., by Adams, Nelson, & Todd, 1992; Davis & Venkatesh, 1996; Mathieson, 1991). The significance of each factor and its variables in determining the acceptance of an innovation have been corroborated by other researches (e.g., Legris, Ingham, & Collerette, 2003; Saadé & Bahli, 2005; Teo, Lim, & Lai, 1999; Venkatesh & Morris, 2000). At the same time, TAM has also widely been adapted without modification or with minor extensions (i.e., by adding other factors that affect users' point of views, e.g., perceived enjoyment, work contexts, etc. or that directly affect users' perceived usefulness, e.g., social influence and cognitive instrumental processes) by many researchers to assess users' acceptance about various

technological innovations. These include tools or software (Babar, Winkler, & Biffi, 2007; Chau, 1996; Davis & Venkatesh, 1996; Laitenberger & Dreyer, 1998) and applications (Henderson & Divett, 2003; Jahangir & Begum, 2008; Pikkarainen, Pikkarainen, Karjaluoto, & Pahnila, 2004; Saadé & Bahli, 2005; Teo et al., 1999).

Table 6.11 TAM Factors and Their Variables

Factor	Variable
Usefulness	1. Work more quickly 2. Job performance 3. Increase productivity 4. Effectiveness 5. Makes Job Easier 6. Useful
Ease of use	1. Easy to learn 2. Controllable 3. Clear and understandable 4. Flexible 5. Easy to become skillful 6. Easy to use

6.3.2 Experiment Participants

Our participants were volunteer students at Universiti Utara Malaysia, Malaysia (http://www.uum.edu.my) who enrolled for the *Computer Modelling in Business* course. This course focuses on the concepts and analyses of DES and uses Arena (Kelton et al., 2004; Kelton et al., 2010) as the implementation software. It is a compulsory course for the students of the *Bachelor of Decision Science* programme and can be taken in the second or third year of the programme. However, other programme students can enrol it as an elective course.

Arena is DES software that uses the SIMAN language (C. Dennis Pegden, Shannon, & Sadowski, 1995) as its simulation engine. DES models are created using modules and connectors to represent their processes and logic. Animation that shows the models' behaviour can be provided using its animation tools. Simulation outputs will automatically be displayed when simulation ends. Figure 6.4 shows a screenshot of Arena.

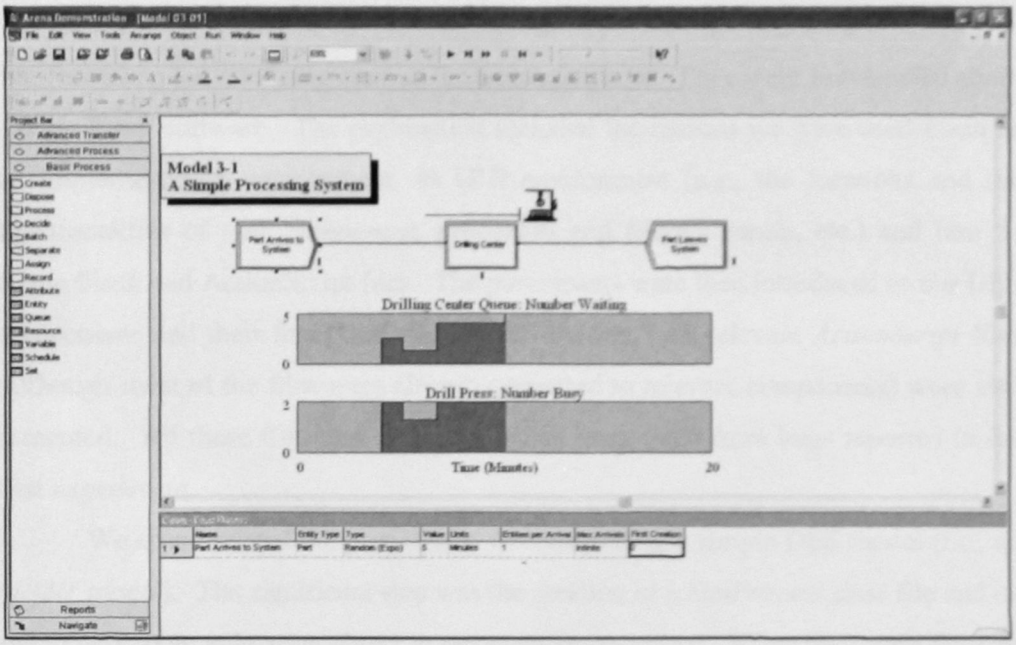


Figure 6.4 Arena Screenshot

We intentionally chose these students since they had been equipped with knowledge on DES concepts and had experiences in using Arena for DES model development. This choice was made since participants with tacit knowledge and experiences of particular contexts can effectively evaluate a tool since they exactly know what they and other users want (Davis & Venkatesh, 1996; Whitworth, Banuls, Sylla, & Mahinda, 2008). The effect of experiences and job relevance on users' perceived usefulness and perceived ease of use, and eventually on their acceptance of a tool has well been documented (e.g., in Venkatesh & Davis, 2000; Whitworth et al., 2008).

40 students participated in this experiment. Besides their knowledge on DES, the participants also had knowledge on programming, particularly on Visual Basic (Harvey M. Deitel, 2006; Zak, 2009) that they learned in the first year of their programmes. We collected their own assessments of their knowledge on DES and programming so that we could properly assess their perceptions of our tool.

6.3.3 Running the Experiment

We first provided a training session for the participants. They were first briefed about Adobe Flash software. The explanation included the reasons we have used Flash as an implementation environment, its GUI environment (e.g., the locations and the functionalities of *tool*, *component*, *properties* and *library* panels, etc.) and how to create Flash and ActionScript files. The participants were then introduced to our DES components and their functionalities in DES models. All relevant *ActionScript* files (although most of the files were already converted to relevant components) were also presented. All these files and components had been fixed from bugs reported in the first experiment.

We then assisted the participants in constructing a simple DES model (i.e., an *M/M/1* model). The significant step was the creation of a *SimProcess* class file and its attachment to an animation object to represent entity arrival. When they were familiar with the model construction processes, they were asked to either add complexity to the model or create a new model of their own. During model building, we were available to answer their questions and were ready to guide them whenever they were stuck. After experiencing with various components for an hour, they were asked to fill out the questionnaire.

As stated earlier, users' experiences can influence their perceived usefulness, perceived ease of use and perceived enjoyment of a tool and eventually affect their acceptance of the tool. Thus, our questionnaire first collected their perceived knowledge on DES concepts, experiences in programming and familiarity with Adobe Flash and its environment.

Items for measuring the perceived usefulness and the perceived ease of use of our tool are shown in Table 6.12. Note that we modified the *work* and *job* keywords in the original items in Davis (1989) and replaced them with *construct* words (see the complete questionnaire in Appendix D). We also included one more factor, *perceived enjoyment*, which has been claimed (e.g., by Pikkarainen et al., 2004; Saadé & Bahli, 2005; Teo et al., 1999) to influence users' acceptance of a tool (denoted as *Perceived Enjoyment* in the questionnaire). All items under these three factors used a five-point Likert-scale that asked the participants to indicate their disagreement or agreement about the items from (1) strongly disagree to (5) strongly agree.

Table 6.12 Items of Perceived Ease of Use, Perceived Usefulness, Perceived Enjoyment and Self-predicted Future Usage of the Component-based Tool

Perceived Usefulness (PU)
PU1: The component-based tool enables me to *construct* DES models that help learn and understand DES concepts *more quickly*.
PU2: The component-based tool improves my *construction performance* on DES models.
PU3: The component-based tool *increases my productivity* of constructing DES models.
PU4: The component-based tool enhances my *effectiveness* of constructing DES models.
PU5: The component-based tool *makes the construction* of DES models *easier*.
PU6: Overall, the component based tool is *useful* for constructing DES models.
Perceived Ease of Use (PEU)
PEU1: Learning to use the component-based tool is *easy* for me.
PEU2: I find the *processes* of using the component-based tool were controllable (clear, understandable and straight forward).
PEU3: My interaction with the component-based tool is *clear and understandable*.
PEU4: I find the component-based tool to be *flexible* to interact with.
PEU5: It is *easy to become skillful* at using the component-based tool.
PEU6: Overall, the component-based tool is *easy to use*.
Perceived Enjoyment (PE):
PE1: I have *fun* interacting with the component-based tool.
PE2: I *enjoy* using the component-based tool.
Self-Predicted Future Usage (SP):
SP1: I intend to *use* the component-based tool to construct DES models in the future
SP2: I intend to *show* others this component-based tool.

Based on the participants' responses, we performed two tests. First, we assessed the reliability of the items in the questionnaire. Second, we evaluated model builders' perceptions on our component-based tool. High responses for the three factors would imply that the tool was useful, easy to use and enjoy to be used.

6.3.4 Data Analysis and Results

6.3.4.1 General Information

Table 6.13 shows the number and the percentage of the participants grouped by their gender. 10.00% of the participants were male while 90.00% were female. Data also revealed that most of the participants were between 20 to 24 years old.

Table 6.13 The Participants' Gender

Gender	N	Percentage
Male	4	10.00%
Female	36	90.00%

As mentioned earlier, relevant knowledge and experiences could influence the participants' cognitive responses (i.e., their perceived usefulness, perceived ease of use, perceived enjoyment, etc.) about the tool (Davis & Venkatesh, 1996; Stoel & Lee, 2003; Taylor & Todd, 1995). Table 6.14 reports how the participants rated their knowledge on DES, their experiences in programming and their familiarity with Adobe Flash and its environment.

Table 6.14 The Participants' Knowledge and Experiences

Experience	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	Mode	Mean	Std. Deviation
DES	1 (2.50%)	8 (20.00%)	23 (57.50%)	8 (20.00%)	0 (0.00%)	3	2.95	0.714
Programming	3 (7.50%)	10 (25.00%)	15 (37.50%)	12 (30.00%)	0 (0.00%)	3	2.90	0.928
Adobe Flash	2 (5.00%)	11 (27.50%)	17 (42.50%)	10 (25.00%)	0 (0.00%)	3	2.88	0.853

The data revealed that only 22.50% of the participants perceived that that they did not have good knowledge on DES. We can also see that 32.50% of the participants disagreed/strongly disagreed that they had good programming experiences and were familiar with Adobe Flash and its environment, respectively. Of the 40 participants, only 25.00% of them perceived that they were familiar with

Adobe Flash. However, most of them stated that they used Adobe Flash to only create a simple animation with little or no ActionScript programming.

6.3.4.2 Questionnaire Reliability and Validity

Based on the participants' feedback, we first measured the reliability of the items in the questionnaire. For this, we conducted a Cronbach's alpha test. Table 6.15 reports the Cronbach's alpha values for perceived usefulness, perceived ease of use and perceived enjoyment factors. All factors showed values higher than 0.8 (the overall reliability was 0.927). Thus, perceived usefulness, perceived ease of use and perceived enjoyment scales showed high levels of reliability (George & Mallery, 2009). This indicates that the questionnaire is a reliable measurement instrument.

Table 6.15 Cronbach's Alpha Values

Factor	Cronbach's Alpa Value
Perceived Usefulness	0.933
Perceived Ease of use	0.890
Perceived Enjoyment	0.823

We also checked the factorial validity of the questionnaire; i.e., whether perceived usefulness, perceived ease of use and perceived enjoyment form distinct constructs. For this, we performed factor analysis with *varimax rotation* that checks which items tend to cluster together. Table 6.16 shows the factor analysis results.

Each value in the Table 6.16 shows the correlation of the variable with the three factors respectively. This value is called a *variable's loading factor*. It can range between -1 (a perfect negative association with the factor) and 1 (a perfect positive association with the factor). A value that closes to 0 indicates that there is no relationship between the variable and the factor. A loading factor of at least 0.7 shows a strong correlation of a variable with a considered factor (J. O. Kim & Mueller, 1978). However, a lower value of 0.5 is sometimes considered important for the factor (Coakes, 2007).

Table 6.16 Factor Analysis of Perceived Usefulness, Perceived Ease of Use and Perceived Enjoyment

Variable	Factor		
	Usefulness	Ease of use	Enjoyment
Work more quickly (PU1)	.714	.206	.359
Job performance (PU2)	.772	.390	.174
Increase productivity (PU3)	.873	.235	.058
Effectiveness (PU4)	.896	.154	.136
Makes Job Easier (PU5)	.826	.351	.046
Useful (PU6)	.820	.089	.203
Easy to learn (PEU1)	.236	.761	.130
Controllable (PEU2)	.226	.777	.312
Clear and understandable (PEU3)	.372	.827	.166
Flexible (PEU4)	.261	.649	.492
Easy to become skilful (PEU5)	.119	.777	.141
Easy to use (PEU6)	.241	.368	.649
Fun (PE1)	.115	.271	.864
Enjoy (PE2)	.142	.090	.872

We can see that all variables except two variables loaded greater than 0.7 on one of the factors. The first variable, i.e., Flexible (PEU4) only had a value of 0.649 on the perceived ease of use factor. However, since this variable had a value greater than 0.5 and loaded higher on the perceived ease of use factor than the other two factors, we could attribute this variable to the perceived ease of use factor. The second variable, i.e., easy to use (PEU6) loaded higher on the perceived enjoyment factor (loading factor = 0.649). Data showed that the easy to use variable had strength correlation with the perceived enjoyment factor.

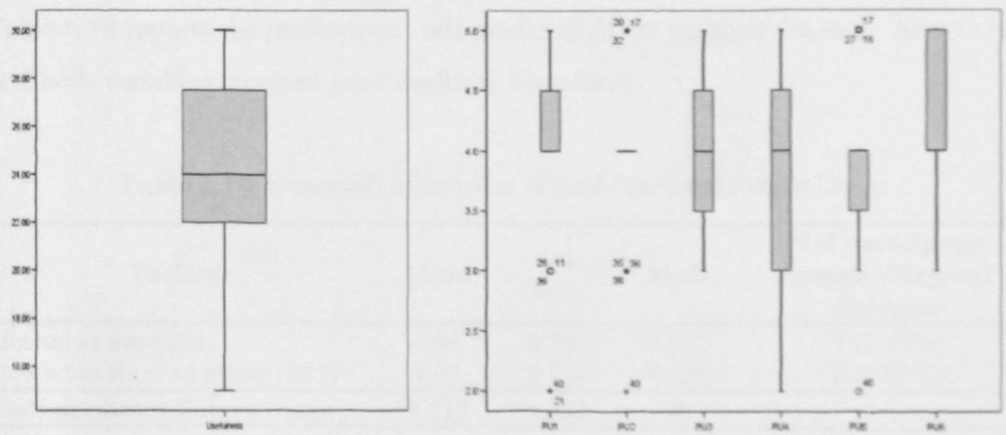
6.3.4.3 Usefulness, Ease of Use and Enjoyment of the Tool

Table 6.17 shows the descriptive statistics for all items in the questionnaire. As we can see, in general, most participants were positive about the tool. Few of the participants (less than 7.50%) disagreed/strongly disagreed with the items related to perceived usefulness, perceived ease of use and perceived enjoyment of the tool (see the last column in Table 6.14).

Table 6.17 Descriptive Statistics of the Items

Variable	Mean	Std. Dev.	Mode	N of Strongly Disagree/Disagree
Work more quickly (PU1)	4.03	0.768	4 (23)	2 (5.00%)
Job performance (PU2)	3.98	0.733	4 (22)	1 (2.50%)
Increase productivity (PU3)	4.00	0.716	4 (20)	0 (0.00%)
Effectiveness (PU4)	3.88	0.822	4 (16)	1 (2.50%)
Makes Job Easier (PU5)	3.93	0.730	4 (22)	1 (2.50%)
Useful (PU6)	4.15	0.700	4 (20)	0 (0.00%)
Perceived Usefulness	**23.95**	**3.876**	**24**	-
Easy to learn (PEU1)	3.65	0.834	4 (19)	3 (7.50%)
Controllable (PEU2)	3.68	0.797	4 (17)	2 (5.00%)
Clear and understandable (PEU3)	3.75	0.840	4 (19)	3 (7.50%)
Flexible (PEU4)	3.83	0.781	4 (18)	1 (2.50%)
Easy to become skillful (PEU5)	3.85	0.700	4 (23)	1 (2.50%)
Easy to use (PEU6)	3.75	0.809	4 (21)	3 (7.50%)
Perceived Ease of Use	**22.50**	**3.830**	**22**	-
Fun (PE1)	3.93	0.572	4 (27)	0 (0.00%)
Enjoy (PE2)	4.08	0.526	4 (29)	0 (0.00%)
Perceived Enjoyment	**8.00**	**1.013**	**8.00**	-

Figure 6.5 reports the results of the tool's perceived usefulness in graphical formats. It shows the summative results (Figure 6.5(a)) and the detail results of each item (Figure 6.5(b)) under this factor. The rating of summative results ranged between 15 and 30 with the mean of 23.95. Considering the maximum rating was 30, we could conclude that most of the participants considered the tool were useful for constructing educational DES models. All variables received good scores (mean above 3.88) with the *useful* variable (PU6) received the highest score with the mean of 4.15.



(a) (b)

Figure 6.5 Perceived Usefulness Results

The sum of items under the perceived ease of use factor ranged between 14 and 30, with the mean value of 22.50. This mean value showed the participants perceived the tool was easy to use. A close examination of different items revealed that all items received positive feedback from most of the participants. The *easy to become skilful* variable (PEU5) was rated with the highest value (with the mean of 3.85). This probably indicates that the drag and drop fashion eases model constructions and demands little guidance. Most of the participants also perceived our tool was *flexible* (mean: 3.83) and *easy to become skilful* (mean: 3.85). The two items that received low ratings from the participants were *easy to learn* (PEU1) and *controllable* (PEU2) with the mean values of 3.65 and 3.68 respectively.

The sum of items under the perceived of enjoyment factor ranged between 6 and 10 with the mean value of 8. This indicated that most of the participants enjoyed using the tool. They also stated that they had fun (mean: 3.93) and enjoyed using the tool and its resulting models (mean: 4.08).

6.3.4.4 Self-predicted Future Usage

The participants were requested to predict their future usage of the tool; i.e., whether they will use the tool if it is available in the future. Such self predictions are among the most accurate predictors available for measuring an individual's future behaviour of an innovation (Sheppard, Hartwick, & Warshaw, 1998; Warshaw & Davis, 1985). Table 6.18 reports the participants' self-predicted future usage of the tool. As we can see, both variables received good feedback from them.

Table 6.18 Descriptive Statistics of Self-Predicted Future Usage

Variable	Mean	Std. Dev.	Mode	N of participants strongly disagreed/ disagreed
Intend to use (SP1)	4.08	0.730	4 (22)	1 (2.00%)
Intend to show to others (SP2)	4.05	0.714	4 (20)	0 (0.00%)
Self-predicted Future Usage	**8.125**	**1.381**	**8**	-

According to the *Theory of Reasoned Action* (Ajzen & Fishbein, 1980; Fishbein & Ajzen, 1975), user's perceived usefulness and perceived ease of use are significantly correlated to the acceptance of an innovation. The acceptance has also been proved by other studies (e.g., Pikkarainen et al., 2004; Saadé & Bahli, 2005; Teo et al., 1999) to be influenced by their perceived enjoyment.

To investigate the degree (strength) of relationships between each of these three factors and the participants' acceptance of our tool, we ran a *Pearson correlation* analysis. For this, we correlated the three summative results of the perceived usefulness, perceived ease of use and perceived enjoyment to the summative results of the participants' predicted future usage. Table 6.19 reports the results of the analysis.

Table 6.19 Correlations between Perceived Usefulness, Perceived Ease of Use and Perceived Enjoyment to Self-Predicted Future Usage

		Usefulness	Ease of Use	Enjoyment	Future Usage
Usefulness	Pearson Correlation	1	.594**	.366*	.428**
	Sig. (2-tailed)		.000	.020	.006
Ease of Use	Pearson Correlation	.594**	1	.562**	.298
	Sig. (2-tailed)	.000		.000	.062
Enjoyment	Pearson Correlation	.366*	.562**	1	.605**
	Sig. (2-tailed)	.020	.000		.000
Future Usage	Pearson Correlation	.428**	.298	.605**	1
	Sig. (2-tailed)	.006	.062	.000	

** Correlation is significant at the 0.01 level (2-tailed).
* Correlation is significant at the 0.05 level (2-tailed).

The results showed that each perceived usefulness and perceived enjoyment was positively correlated with self-predicted future usage. This indicated that both of the factors were important determinants influencing the participants' future usage of the tool. The correlation coefficient between perceived enjoyment and self-predicted future usage was much higher than the correlation coefficient between perceived usefulness and self-predicted future usage; i.e., 0.605 ($p < 0.005$) compared to 0.428 ($p < 0.010$). However, we are not confident that there was a correlation between perceived ease of use and self predicted future usage since the p-value was greater than 0.05. This hints that the participants opted to use the tool primarily because of its usefulness and perceived enjoyment compared to its ease of use. We can also see that

there was a correlation between usefulness and ease of use ($r = 0.594$, $p < 0.005$), usefulness and enjoyment ($r = 0.366$, $p < 0.010$) and ease of use and enjoyment ($r = 0.562$, $p < 0.005$).

To reveal predictive power between self-predicted usage of the tool and the three individual factors, regression analyses were conducted. Table 6.20 shows the regression analysis results. The results clearly showed that perceived usefulness and perceived enjoyment had positive effects on self-predicted future usage.

Table 6.20 Regression Analyses of the Effect of Perceived Usefulness and Perceived Ease of Use on Self-Predicted Future Usage

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	.671(a)	.451	.405	1.06543

Predictors: (Constant), EaseOfUse, Enjoy, Usefulness

Coefficients

Model		Unstandardized Coefficients		Standardized Coefficients	t	Sig.
		B	Std. Error	Beta		
1	(Constant)	.429	1.486		.289	.775
	Usefulness	.127	.055	.356	2.317	.026
	Enjoyment	.849	.204	.623	4.165	.000
	Ease of Use	-.095	.062	-.264	-1.523	.136

Dependent Variable: Future Usage

The R^2 of the regression was 0.451. However, the R^2 value is generally of secondary importance unless the regression model will be used to make accurate predictions. To tell how confidence we are that each of the independent variables (i.e., perceived usefulness, perceived ease of use and perceived enjoyment) has some correlation with the dependant variable (i.e., future usage), we should observe the p-values of each variable.

The p-values for perceived usefulness ($p = 0.026$) and perceived enjoyment ($p = 0.000$) were smaller than 0.05. This indicated that both of the factors were useful predicators for self predicted future usage. The analysis also revealed that perceived enjoyment was the most influential factor to self-predicted usage ($t = 4.165$, $p < 0.05$). Note that the t-value suggests the relative importance of each variable in the model

and t-value between -2 and 2 reflects a useful predicator. Our finding of perceived enjoyment has significant effect on an innovation is tally with some researchers' finding (e.g., by Pikkarainen et al., 2004; Saadé & Bahli, 2005; Teo et al., 1999). However, this finding is in contrast with that of other researchers (e.g., Igbaria, Livari, & Maragahh, 1995) that claimed perceived enjoyment was not related to the acceptance of an innovation.

Perceived usefulness was also found to be the influential factor to self-predicted usage ($t = 2.317$, $p < 0.005$). This finding is in line with other TAM studies (e.g., Davis, 1989; Davis, Bagozzi, & Warshaw, 1989; Igbaria, Zinatelli, Cragg, & Cavaye, 1997; Keil, Beranek, & Konsynski, 1995) that found perceived usefulness had more impact on technology acceptance than perceived ease of use since perceived ease of use impinges on acceptance through perceived of usefulness. However, some researchers claim the opposite (Chau, 1996; Venkatesh & Morris, 2000). We can see that the ease of use factor had small relation with the self predicted future usage as indicated by its non-significant t-value ($p > 0.05$).

6.3.4.5 Participants' Cognitive Workloads

It is important to measure the participants' cognitive workloads while using our tool. There are two approaches for measuring this: Short Subjective Instrument (SSI) (Paas, Tuovinen, Tabbers, & Gerven, 2003) and the NASA TLX mental workload instrument (Hart, Stavenland, Hancock, & Meshkati, 1988).

The SSI assesses a participant's overall cognitive workload using a single question that requests him/her to rate a given task from extremely easy (1) to extremely difficult (7). We chose the NASA TLX since it can assess the level of the participant's various cognitive loads, based on the combination of his/her *extraneous load* (i.e., his/her memory load while using a material but this can be controlled by the material's designer; e.g., through the use of graphics or relevant presentation formats) and *intrinsic load* (i.e., the level of perceived difficulty of a material and this can be influenced by his/her knowledge and experience). Both types of workloads are measured using the NASA TLX instruments based on six factors:

- Mental demand; i.e., if the task affects the participant's attention
- Physical demand; i.e., if the task affects the participant's health
- Temporal demand; i.e., if the task consumes a lot of time that the participant cannot afford
- Performance; i.e., if the task is heavy or light in terms of workload
- Frustration; i.e., if the task makes the participant unhappy
- Effort; i.e., if the participant has spent a lot of effort on the task

In our case, the physical demand factor reflects the participants' physical tension and stress while and after developing DES models using our tool. We have explained this term to them during the experiment. The more they use keyboards and mice, the more physical activities they have to perform and these may cause pain in the back, neck, shoulder and muscle, strain on the eyes and strain on fingers, etc. We would like them to rate how the activities affected their health.

Originally, the NASA TLX calculates the participants' overall cognitive workloads based on their responses to pair-wise comparisons among the six factors and their ratings on each of these factors. However, the factor rating is the most important element in calculating the overall workload score; and removing the pair-wise comparisons may increase the experimental validity and reduce the experimental error (Bustamante, 2008). Since our purpose was to generally assess our participants' cognitive workloads while using our tool, we only requested them to rate the six factors based on a 7-point scale (1 = low, 7 = high). Table 6.21 shows their feedback for each of the factors. The overall cognitive workload for all of the participants were close to average with mean = 3.642 and standard deviation = 1.104. This value indicated that the participant's mental requirement for building DES sample models using our tool was not so simple since they had to do some hands-on tasks (e.g., creating class files, attaching the files to their relevant objects, dragging, dropping and connecting the components, etc.) and was not so complex since they had been equipped with knowledge on and experiences in the domain.

Table 6.21 Participants' Feedback about the TLX Subscales

Factor	1	2	3	4	5	6	7	Mode	Mean	SD
Mental demand	2 (5.00%)	8 (20.00%)	7 (7.50%)	4 (10.00%)	11 (27.50%)	7 (17.50%)	1 (2.50%)	5	3.98	1.641
Physical demand	2 (5.00%)	7 (17.50%)	11 (27.50%)	9 (22.50%)	8 (20.00%)	3 (7.50%)	0 (0.00%)	3	3.58	1.338
Temporal demand	0 (0.00%)	7 (17.50%)	6 (15.00%)	16 (40.00%)	6 (15.00%)	4 (10.00%)	1 (2.50%)	4	3.93	1.289
Performance	2 (5.00%)	10 (25.00%)	13 (32.50%)	9 (22.50%)	5 (12.50%)	1 (2.50%)	0 (0.00%)	3	3.20	1.181
Frustration	2 (5.00%)	9 (22.50%)	4 (10.00%)	7 (17.50%)	13 (32.50)	5 (12.50%)	0 (0.00%)	5	3.88	1.522
Effort	1 (2.50%)	8 (20.00%)	18 (45.00%)	5 (12.50%)	7 (17.50%)	1 (2.50%)	0 (0.00%)	3	3.30	1.137

There were two main complaints noted by the participants about the tool. First, some of the participants complained that the approach of linking components based on their specified names during design time tended to create logical errors. Many of them experienced this. These hard-to-trace errors happened when specified downstream component names were misspelled in their upstream component's *outport* property. As a result, entity flows to the upstream components would be broken. They suggested that the components should easily be conneeted during design time; e.g., using arrows. Secondly, the requirement processes of creating an entity class file and attaching it to an animation object really burdened them and should be simplified. We explained that we could actually create a library that consists of various considered entities. However, permitting model builders to define and create their own entities would give flexibility for them in animating the entities.

The analyses of various feedback in the first experiment confirmed that our component-based tools produced *attractive*, *interactive* and *informative* DES models which were suitable for learning and teaching purposes. Its attractiveness in terms of animations (e.g., high quality animated objects and events, different images of objects based on their states, etc.) makes learning enjoyable and fun. Its interactiveness in relation to permitting learners to manipulate the models' parameters through easy-to-access GUIs, controlling the speed of simulation and customizing the models' visualizations by adding, removing and relocating relevant data visualizations (e.g., graphs, tables, etc.) to any locations during runtime helps learners to understand the model's behaviour. Its informative feature that provides feedback on the impact of

parameter changing through various meaningful animations and animated data visualizations aids learners to clarify their ideas and understand various scenarios in the models. The analyses of various feedback in the second experiment reflected that our DES components were useful, easy to use and enjoy to be used to build these kinds of models. However, there is still a room for their future improvements. These include investigating how to easily link the components instead of typing the names of their downstream components in a layout property and providing various libraries of entities and resources for model developers to easily animate the objects without the need to create their appropriate classes.

CHAPTER 7

CONCLUSION AND FUTURE RESEARCH

7.1 Introduction

This chapter consists of three sections: *Conclusions*, *Limitations of the Research* and *Recommendations for Future Research*. The Conclusions section summarizes and discusses the findings of this research. The Limitations of the Research section lists and discusses some weaknesses of this research. The Recommendations for Future Research section proposes some ideas for future research.

7.2 Conclusion

Many studies have expressed strong support for the use of games and simulations as educational tools. Their support is mainly based on the hypothesis that learners implicitly acquire target knowledge during their engagement and interactions with the models. Although such interactions can create different motivation levels of learning (e.g., learners with good mental models of a domain may lose interest since its model's outputs can well be predicted, while other learners with less detailed models may lose motivation since the outputs induces no significance cognitive responses), many educators believe that the right design of a model can facilitate learning. Examining the benefits of using various types of simulation models and dealing with their potential constraints in the learning and teaching environment were one of the research contributions.

The main contribution of this research is the proposal of how to construct DES tools for building *attractive*, *interactive* and *informative* DES models to be used as learning and teaching materials. Before this work, DES was typically used as an analysis tool for system performance prediction and its outputs were only usable for

system modellers. Thus, in order to offer the benefits of DES to the education community especially in helping learners understand the effect of capacity constraints on the performance of a system, we proposed a component-based tool approach. This approach allows DES tool developers to directly embed the three model features suggested in literature review (i.e., hypothesis test platforms, concurrent responsive animations and customized visualizations) that help to get insight into DES behaviour during their learning through models.

The use of *attractive* and *interactive* models of soft skill simulations or procedural simulations to support basic concepts of relevant theories is common in educational settings. In fact, these types of models can easily be constructed even without using a commercial tool, since the rules regulating their logic are fully structured. However, educational models of open-ended simulations (i.e., DES that analyses a system's performances or continuous simulations that deal with complex natural processes) that allow learners to get insight into most of real world systems are uncommon. One reason for this is that their operations involve a lot of computation that hinders model builders from constructing their own models without the help of the right tool.

Current DES tools have some distinct weaknesses. Most of the free research tools are not easy to use since model construction requires a lot of programming and their resulting models offer no animation and visualized structures. Better research tools, although supporting model construction through a drag and drop fashion to a certain extent, do not typically integrate good animation and visualization capabilities. Commercial tools provide high quality animation and visualization. However, the tools restrict further extension. Their resulting models must also be played using the software's player and this hinders the models from being accessed through internet or integrated with LMSs. Additionally, no single tool generates models with *runtime interactions* and *visualization customization* capabilities; i.e., two important characteristics that facilitate learning according to many educational studies.

This thesis focuses on designing and developing a DES tool to help model builders to construct *educational* DES models. These models facilitate learners to get insight into DES concepts through model interactions, customized state visualization, entities' and resources' animation and animation speed manipulation during runtime. Model interactions help learners to perform *what-if* experiments without the need to modify models' source code. Customized state visualizations ease them to control the

amount of information displayed on computer screen at a time; i.e., each learner can construct his/her own model GUIs by adding or removing particular state visualization during runtime. Additionally, the GUIs and any interesting scenario (i.e., its current models' states, animation and visualization) can be saved at any time to be loaded in the future. Speed manipulation gives flexibility to learners to look closer at aspects that catch their attention and skip over aspects that are of no current interest. Furthermore, arrows that depict paths of entity movements for viewing various entities' lifecycles that would be helpful for more complex models are also supported. These features are important in the learning environment, but often neglected in the current DES tools, since their main focus is on system performance analyses.

To systematically design such a tool, we first architected a *framework* that consists of classes with their own functionalities. We have shown that this framework was *flexible* enough to support the construction of various queuing models and their specific logics, and *extensible* to cater various types of DES models. Model construction tasks have now been relieved from the many of the routine tasks associated with DES models using an object-oriented style that supports the concepts of *inheritance*, *encapsulation* and *polymorphism*. However, the model building is only through Application Programming Interface (API); i.e., an amount of programming that uses to show relationships between objects of the classes is still needed to represent their logics.

To support the tool's *ease of use* feature through a component drag and drop fashion and to ensure that its resulting models are *informative, useful and enjoyable* to be used in the learning and teaching environment, we proposed the combination of two design patterns; i.e., the *Delegation Event Model (DEM)* which is used to link the models' components together, and the *Model-View-Controller (MVC)* pattern which is used to support their GUIs and customisable visualizations during runtime. Implementing the DEM pattern in the DES components allows us to flexibly specify various entities' lifecycles during design time without the need to write conditional statements, while implementing the MVC pattern allows us to freely link various visualization tools with the components without the need to refer them in the components' code. Thus, various visualization facilities that render generated data during simulation can be automated or integrated with ease. We later showed how a component's states and its relevant animation and visualization can be saved for

future use. How these two design patterns support the development of a hierarchical simulation model (i.e., how to connect and synchronize the model with its children so that entities can be transferred between layers in the right orders) has also been architected and discussed in detail.

We used Adobe Flash as the tool's implementation language for two reasons. First, it expedites the development processes of the components; e.g., through its layout properties, facilities to attach objects with their classes and animate them based on their states, stage for composing the components, etc. Second, it automatically generates web-based and LMS-compatible models. With the right design and environment, we believe that our tool eases the construction of useful DES models.

As mentioned earlier, we designed and constructed DES tools to effectively support three groups of users; i.e., developers, teachers and learners. We did not investigate how easily developers could expend the tools to support other DES applications; e.g., manufacturing, logistic, etc. However, we believed the tools could easily be extended since their development are based on *UML* (Unified Modelling Language) class diagrams (that clearly shows its relevant classes, methods attributes and the relationships among the classes) and two well-known designed patterns, i.e., the *Delegation Event Model* and the *Model-View-Controller* which are common approaches to all software developers. We however investigated the feedback from teachers about the tools' usefulness and the ease of use and learners about the tools' attractiveness and interactivity through experiments.

Perceived *usefulness*, perceived *ease of use* and perceived *enjoyment* have been claimed as crucial factors that determine the acceptance of a tool. To assess if our component-based tool and its resulting models support these three factors, we conducted two experiments. The first experiment basically evaluated if the tool's resulting models were attractive, interactive, informative and useful enough to be used for learning and understanding DES concepts. The results of the experiment showed that a majority of the 28 participants gave positive feedback for all items in our questionnaire. The items were constructed based on essential model features claimed by previous studies. The second experiment assessed usefulness, ease of use and enjoyment of the tool from model builders' perspectives; i.e., their experiences while using the tool to construct DES models. Items for measuring these factors were designed based on the Technology Acceptance Model (TAM) and other previous relevant studies. Participants were from those that had knowledge on DES and

programming. Analyses of their feedback showed that a majority of the 40 participants found that the tool was useful, easy to use and enjoyable. They were also very positive about the regular use of the tool for constructing DES models in the future.

The feedback analyses of the second experiment also revealed that perceived enjoyment and perceived usefulness were important determinants for the tool acceptance. However, perceived enjoyment was discovered to be a critical factor for its acceptance. Perceived ease of use meanwhile was found to have a relatively weak relationship with the participants' acceptance. We also assessed the level of the participants' perceived cognitive workloads while experiencing the tool using the NASA **T**ask **L**oad Inde**X** (TLX) instrument. The results showed that the overall workload for all participants based on a 7-point scale (1 = low, 7 = high) was 3.642 (standard deviation = 1.104); i.e., their mental requirements while using the tool were not too simple and not too complex.

7.3 Limitations of the Research

We only focused on the design and development of DES components for building DES educational models. Each component symbolises the location where relevant events and their occurrence time may take place while their linkages provide visualization structures of various entity flows. This logic can suit many types of real-life systems; e.g., service, transportation and manufacturing systems.

In case of a continuous system where its states change continuously, the ideas of components that simplify its model building and allow exploring its behaviour through various GUIs and visualizations are still relevant. However, representing its operational logic may only need three types of components; i.e., *level* or *stock* that stores variables of continuous processes that are always changing, *rate* or *flow* that defines the rates of change of the variables over time and these rates may depend on other continuous processes, and *setup* (a continuous simulation engine) that configures all continuous simulation calculation (e.g., size of increment time steps, the numerical method to be used, etc.). The linkage between *level* and *rate* components is much simpler since it only involves the assignments of variables with their relevant differential equations that represent the rates of change of the variables.

However, specifying the equations is only possible through an API. This requires model builders to have some basic programming knowledge besides their mental model of a system being constructed.

Our tool's resulting models do not offer model construction capabilities at run time. Right now, learners can only experiment with the models and customize their visualizations. Allowing them to alter the existing model structures or create a new model during runtime may offer some educational benefits especially in facilitating their understanding of various DES aspects from model building to model analyses. This can be achieved through providing a palette that floats around the models during runtime and contains various model construction components, entity and resource objects.

We used the Flash environment and its *ActionScript* as an implementation language for constructing DES components. The use of other languages although possible may introduce additional burdens since they may not provide facilities for simplifying component development (e.g., facilities for attaching an object to a class, embedding default GUIs to the components, etc.) and animation capabilities. However, the design and development techniques that have been discussed in this thesis can be implemented and extended in any other object oriented programming languages.

Other limitations of the research relate to the experiment limitations. Firstly, both of the experiments used small sample sizes of participants. The number of participants in the first experiment was only 28 while the number of participants in the second experiments was 40. Such small sample sizes definitely had an effect on the ability to generalize the findings. As a result, we could not give conclusive evidence about *learners' perceptions* on attractiveness and interactivity of our tool's sample models and *model builders' perceptions* on the usefulness, ease of use and enjoyment of the tool for constructing DES models. However, we believe that these sample sizes were sufficient enough for obtaining and reporting users' feedback about the tool. In order to have greater confidence that the experiment results are representative, we should have a large number of voluntary participants. Secondly, both experiments also suffered from other possible factors; i.e., *social influence processes* that directly affected the participants' acceptance of the tool and *cognitive instrumental processes* that influenced perceived usefulness and perceived ease of use of the tool (see Venkatesh & Davis, 2000).

7.4 Recommendations for Future Research

Continuous systems can be found anywhere in our life; e.g., plant and animal growth, human population, weather changes, etc. However, relevant models that ease learning of their behaviour are uncommon. Current tools not only require an amount of programming code to represent the systems' dynamic processes, but their resulting models do not also allow adjustment of different aspects of their parameters and customization of their visualizations during runtime. In this case, component-based tools may ease the construction of attractive and interactive continuous simulation models. However, how to properly structure such components to continuously track system responses over time according to a set of differential equations and how to support the resulting models' GUIs so that their parameters and relevant equations can be changed on the fly are worth to be investigated. Hopefully, there will be research that will investigate this matter.

Many studies claim that interactions during classroom enhance learning. However, few researchers focus on studying learners' interactions while using an open-ended simulation model for making judgement about their learning. Investigating various factors (e.g., how long they have used the model, how many times they have clicked relevant objects, what model parameters they have changed, what additional evaluation need to embedded in the model, how to judge their understanding, etc.) may signal their learning are worth exploring. This is possible since all relevant data about their interactions while using the model can be captured and analysed (either using LMS facilities or by the model itself). The next step is just to develop mechanisms that relate all the data to induce relevant conclusions about the effective use of the model.

Guiding exploration on open-ended models through a list of structured activities may help learning and decrease their sense of being lost during exploration. For this, the models must have quality and aesthetics values to support various exploration capabilities. Finding a way of how to judge or measure the quality of a model based on educational perspectives and how to better structure more flexible objects that enable learners to deeply drill down their hierarchies (i.e., their internal structures, operations and possibly into their source code) step by step via modal windows is another possibility of a future research. This feature will not only enable learners to visualize and analyse the model (e.g., through its multiple views of

structures, states, abstraction levels, composition, etc.), but also help them to easily understand how important processes and properties of a real system are presented in a computer environment.

Our future work includes upgrading our components to support the proposed hierarchical models discussed in Chapter 5. If they function as outlined, this will be a great enhancement to our component-based simulation tool since the tool now supports both of the construction of attractive and interactive a single layer and multi layer DES models.

REFERENCES

Aamodt, A., & Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications, 7*(1), 39-59.

Adams, D.A., Nelson, R.R., & Todd, P.A. (1992). Perceived Usefulness, Ease of Use, and Usage of Information Technology: A Replication. *MIS Quarterly, 16*(2), 227-247.

Ainsworth, S. (1999). The Functions of Multiple Representations. *Computers & Education, 33*, 131-152.

Ainsworth, S., Bibby, P., & Wood, D. (2002). Examining the Effects of Different Multiple Representational Systems in Learning Primary Mathematics. *Journal of the Learning Sciences, 11*(1), 25 - 61.

Ajzen, I., & Fishbein, M. (1980). *Understanding Attitudes and Predicting Social Behavior.* Englewood Cliffs, NJ: Prentice Hall.

Ala-Mutka, K., Gaspar, P., Kismihok, G., Suurna, M., & Vehovar, V. (2010). Status and Developments of eLearning in the EU10 Member States: The Cases of Estonia, Hungary and Slovenia. *European Journal of Education, 45*(3), 494-513. doi: 10.1111/j.1465-3435.2010.01442.x

Alam, G.M., Oloruntegbe, O.K., Oluwatelure, A.T., Alake, M., & Ayeni, A.E. (2010). Is 3D just an Addition of 1 to 2 or Is It More Enhancing Than 2D Visualizations. *Scientific Research and Essays, 5*(12), 1536–1539.

Aldrich, C. (2002). A Field Guide to Educational Simulations. Retrieved Oct 18, 2007, from http://www.simulearn.net/pdf/astd.pdf

Aldrich, C. (2004). *Simulations and the Future of Learning: An Innovative (and Perhaps Revolutionary) Approach to e-Learning.* San Francisco, California: Pfeiffer.

Aldrich, C. (2005). *Learning by Doing: A Comprehensive Guide to Simulations, Computer Games, and Pedagogy in e-Learning and Other Educational Experiences.* San Francisco, California: Pfeiffer.

Alejandra, C., Mario, P., & Antonio, V. (2003). *Component-Based Software Quality: Methods and Techniques.* Berlin: Springer.

Alonso, F., Lopez, G., Manrique, D., & Vies, J.M. (2005). An Instructional Model for Web-based e-learning Education with a Blended Learning Process Approach. *British Journal of Educational Technology, 36*(2), 217-235.

Anderson, J.R., Corbett, A.T., Koedinger, K.R., & Pelletier, R. (1995). Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences, 4*(2), 167-207.

Anderson, L.W., & Krathwohl, D.R. (2000). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives.* Boston: Allyn & Bacon.

Aragon, S.R., Johnson, S.D., & Shaik, N. (2002). The Influence of Learning Style Preferences on Student Success in Online Versus Face-to-face Environments. *American Journal of Distance Education, 16*(4), 227-245.

Arbaugh, J.B., & Benbunan-Fich, R. (2007). The Importance of Participant Interaction in Online Environments. *Decision Support Systems, 43*(3), 853-865. doi: http://dx.doi.org/10.1016/j.dss.2006.12.013

Arnold, K., Gosling, J., & Holmes, D. (2006). *The Java Programming Language* (4th ed.). Upper Saddle River: Addison-Wesley.

Atkinson, C., Bunse, C., Gross, H.-G., & Peper, C. (2005). *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*. Berlin: Springer-Verlag.

Au, G., & Paul, R.J. (1996). Visual Interactive Modelling: A Pictorial Simulation Specification System. *European Journal of Operational Research, 91*(1), 14-26.

Aubidy, K.M.A. (2007). Teaching Computer Organization and Architecture Using Simulation and FGPA Applications. *Journal of Computer Science, 3*(8), 624-632.

Babar, M.A., Winkler, D., & Biffi, S. (2007). Evaluating the Usefulness and Ease of Use of a Groupware Tool for the Software Architecture Evaluation Process. *First International Symposium on Empirical Software Engineering and Measurement 2007 (ESEM 2007)*, 430-439.

Banduras, A. (1977). Self-efficacy: Toward a Unifying Theory of Behavioral Change. *Psychological Review, 84*(2), 191-215.

Banks, J. (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. New York: John Wiley & Sons.

Bapat, V., & Sturrock, D.T. (2003). The Arena Product Family: Enterprise Modeling Solutions. *Proceedings of the 2003 Winter Simulation Conference*, 210-217.

Barnes, C.D., & Laughery, J.K.R. (1997). Advanced Uses for Micro Saint Simulation Software. *Proceedings of the 1997 Winter Simulation Conference*, 680-686.

Bedor, H.S., Mohamed, H.K., & Shedeed, R.A. (2004). A General Architecture of Student Model to Assess the Learning Performance in Intelligent Tutoring Systems. *Proceedings of International Conference on Electrical, Electronic and Computer Engineering 2004*, 173- 178.

Belfore, A.L., Mielke, R.R., & Kunam, K.C. (2003). A Framework for Creating VRML Visualizations from Discrete Event Simulations. *Proceedings of the International Symposium on Collaborative Technologies and Systems*, 93-98.

Bell, P.C. (1989). Stochastic Visual Interactive Simulation Models. *Journal of the Operational Research Society, 40*, 615–624.

Benjamin, D.M., Mazziotti, B.W., & Armstrong, F.B. (1994). Issues and Requirement for Building a Generic Animation. *Proceedings of the 1994 Winter Simulation Conference*, 1304-1310.

Beux, P.L., & Fieschi, M. (2007). Virtual Biomedical Universities and e-learning. *International Journal of Medical Informatics, 76*(5-6), 331-335.

Birtwistle, G.M. (1979). *DEMOS: A Discrete Event Modelling on Simulation*. London: McMillan.

Birtwistle, G.M. (1980). *Simula Begin* (2 ed.). Lund, Sweden: Studentlitteratur.

Bodemer, D., & Faust, U. (2006). External and Mental Referencing of Multiple Representations. *Computers in Human Behavior, 22*(1), 27-42.

Bose, S.K. (2002). *An Introduction to Queueing Systems*. New York: Kluwer Academic/Plenum Publisher.

Boyar, J. (1989). Inferring Sequences Produced by Pseudo-random Number Generators. *Journal of the ACM (JACM), 36*(1), 129 - 141

Bransford, J.D. (2000). *How People Learn: Brain, Mind, Experience and School*. Washington, D.C: National Academy Press.

Brouwer, N., Muller, G., & Rietdijk, H. (2007). Educational Designing with MicroWorlds. *Journal of Technology and Teacher Education, 15*(4), 439-462.

Browne, T., Jenkins, M., & Walker, R. (2006). A Longitudinal Perspective Regarding the Use of VLEs by Higher Education Institutions in the United Kingdom. *Interactive Learning Environments, 14*(2), 177-192.

Bryant, R.M. (1981). A Tutorial on Simulation Programming with SIMPAS. *Proceedings of the 1981 Winter Simulation*, 363-377.

Bunt, A., Conati, C., Huggett, M., & Muldner, K. (2001). On Improving the Effectiveness of Open Learning Environments through Tailored Support for Exploration. *Proceedings of the 10th International Conference on Artificial Intelligence in Education (AI-ED 2001)*, 365-376.

Bunt, A., Conati, C., & Muldner, K. (2004). Scaffolding Self-explanation to Improve Learning in Exploratory Learning Environments. *Intelligent Tutoring Systems, 3220*, 656-667.

Buss, A. (2000). Component-Based Simulation Modelling. *Proceedings of the 2000 Winter Simulation Conference*, 964-971.

Buss, A. (2002). Component Based Simulation Modeling with SIMKIT. *Proceedings of the 2002 Winter Simulation Conference*, 243-249.

Buss, A., & Blais, C. (2007). Composability and Component-Based Discrete Event Simulation. *Proceedings of the 2007 Winter Simulation Conference*, 694-702.

Bustamante, E.A., & Spain, R. D. . (2008). Measurement Invariance of the NASA TLX. *Human Factors and Ergonomics 52*, 1522-1526.

Castagna, G. (1997). *Object Oriented Programming: A Unified Foundation*. Boston: Birkhauser.

Castillo, S., Hancock, S., & Hess, G. (2004). *Using Flash MX to Create e-Learning* (1st ed.). Vancouver: Rapid Intake Press.

Chang, K.-E., Chen, Y.-L., Lin, H.-Y., & Sung, Y.-T. (2008). Effects of Learning Support in Simulation-based Physics Learning. *Computers & Education, 51*(4), 1486-1498.

Charles, C.M. (2008). *Today's Best Classroom Management Strategies: Paths to Positive Discipline*. Boston: Pearson/Allyn Bacon.

Chau, P.Y.K. (1996). An Empirical Investigation on Factors Affecting the Acceptance of CASE by Systems Developers. *Information and Management, 30*, 269-280.

Chen, G., & Szymanski, B.K. (2002). COST: A Component-Oriented Discrete Event Simulator. *Proceedings of the 2002 Winter Simulation Conference*, 776-782.

Cho, Y.I., & Kim, T.G. (2002). DEVS Framework for Component-based Modeling/Simulation of Discrete Event Systems. *Proceedings of the 2002 Summer Computer Simulation Conference*.

Chwif, L., & Barretto, M.R.P. (2003). Simulation Models as an Aid for the Teaching and Learning Process in Operations Management. *Proceedings of the 2003 Winter Simulation Conference*, 1994-2000.

Clark, R.C., Nguyen, F., & Swelle, J. (2006). *Efficiency in Learning: Evidence-based Guidelines to Manage Cognitive Load*. San Francisco: Jossey-Bass.

Clark, R.E., Yates, K., Early, S., & Moulton, K. (2010). An Analysis of the Failure of Electronic Media and Discovery-Based Learning. In K. H. Silber & W. R. Foshay (Eds.), *Handbook of Improving Performance in the Workplace: Volumes 1* (pp. 263-297). San Francisco: Pfeiffer.

Coakes, S.J. (2007). *SPSS Version 12.0 for WIndows: Analysis without Anguish*. Singapore: John Wiley & Sons Australia.

Concannon, K., Elder, M., Hindle, K., Tremble, J., & Tse, S. (2006). *Simulation Modeling with SIMUL8*. Mississauga, Ontario: Visual Thinking International.

Conway, R., & Maxwell, W. (1987). *Modeling Asynchronous Materials Handling Systems in XCELL+*. Paper presented at the Proceedings of the 19th Conference on Winter Simulation.

Craig, I.D. (2007). *The Interpretation of Object Oriented Programming Languages*. London: Springer.

Crain, R.C., & Henriksen, J.O. (1999). Simulation Using GPSS/H. *Proceedings of the 1999 Winter Simulation Conference*, 182-187.

Cronbach, L. (1951). Coefficient Alpha and the Internal Structure of Tests. *Psychometrika, 16*(3), 297-334. doi: 10.1007/bf02310555

Davies, C., H., J. (2002). Student Engagement with Simulations. *Computers and Education, 39* (3), 271-282.

Davis, F.D. (1989). Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly, 13*(3), 319-340.

Davis, F.D., Bagozzi, R.P., & Warshaw, P.R. (1989). User Acceptance of Computer Technology: A Comparison of Two Theoretical Models. *Management Science, 35*(8), 982-1003. doi: 10.1287/mnsc.35.8.982

Davis, F.D., & Venkatesh, V. (1996). A Critical Assessment of Potential Measurement Biases in the Technology Acceptance Model: Three Experiments. *International Journal of Human-Computer Studies, 45*(1), 19-45.

Deitel, H.M. (2006). *Visual Basic 2005: How to Program*. Upper Saddle River: Pearson Prentice Hall.

Deitel, H.M., Deitel, P.J., & Goldberg, A.B. (2004). *Internet & World Wide Web: How to Program* (3rd ed.). New Jersey: Pearson Education International.

Djajadiningrat, T., Matthews, B., & Stienstra, M. (2007). Easy Doesn't Do It: Skill and Expression in Tangible Aesthetics. *Personal Ubiquitous Computing, 11*(8), 657-676. doi: 10.1007/s00779-006-0137-9

Dochy, F., Segers, M., & Buehl, M.M. (1999). The Relation between Assessment Practices and Outcomes of Studies: The Case of Research on Prior Knowledge. *Review of Educational Research, 69*(2), 145-186.

Donatis, A.D. (2006). *Advanced ActionScript Components: Mastering the Flash Component Architecture*. Berkeley: APress.

Donikian, S., & Cozot, R. (1995). General Animation and Simulation Platform. *Computer Animation and Simulation '95*, 197-209. .

Dublin, L. (2004). The Nine Myths of e-learning Implementation: Ensuring the Real Return on Your e-learning Investment. *Industrial and Commercial Training, 36*(7), 291-294.

Duinkerken, M.B., Ottjes, J.A., & Lodewijks, G. (2002). The Application of Distributed Simulation in Tomas: Redesigning a Complex Transportation Model. *Proceedings of the 2002 Winter Simulation Conference*, 1207-1213.

Ebner, M., & Taraghi, B. (2010). Personal Learning Environment for Higher Education – A First Prototype. *World Conference on Educational Multimedia, Hypermedia and Telecommunications 2010*, 1158-1166.

Eck, R.V., & Dempsey, J. (2002). The Effect of Competition and Contextualized Advisement on the Transfer of Mathematics Skills in a Computer-Based Instructional Simulation Game. *Educational Technology Research and Development, 50*(3), 23-41.

Eden, A.H. (2002). A Theory of Object-Oriented Design. *Information Systems Frontiers, 4*(4), 379-391.

Eppler, M.J., & Burkhard, R.A. (2007). Visual Representations in Knowledge Management: Framework and Cases. *Journal of Knowledge Management, 11*, 112-122.

Falvo, D.A. (2008). Animations and Simulations for Teaching and Learning Molecular Chemistry. *International Journal of Technology in Teaching and Learning, 4*(1), 68–77.

Falvo, D.A., & Johnson, B.F. (2007). The Use of Learning Management Systems in the United States. *TechTrends, 51*(2), 40-45. doi: 10.1007/s11528-007-0025-9

Fenrich, P. (2006). Getting Practical with Learning Styles in Live and Computer-based Training Settings. *The Journal of Issues in Informing Science and Information Technology, 3*, 233-242.

Filippi, J.B., Delhom, M., & Bernardi, F. (2002). The JDEVS Modelling and Simulation Environment. *Proceedings of the 1st Biennial Meeting of the iEMSs*, 283-288.

Fishbein, M., & Ajzen, I. (1975). *Belief, Attitude, Intention and Behavior: An Introduction to Theory and Research* Massachusetts: Addison-Wesley.

Fishwick, P.A. (1992). SimPack: Getting Started with Simulation Programming in C and C++. *Proceedings of the 1992 Winter Simulation Conference*, 154-162.

Fitzpatrick, S. (2003). A Review of Web-based Learning and Teaching. Retrieved Nov 20, 2008, from http://www.le.ac.uk/cc/rjm1/etutor/elearning/reviewofwebbasedtl.html

Fletcher, J.D., & Tobias, S. (2005). The Multimedia Principle. In R. E. Mayer (Ed.), *Cambridge Handbook of Multimedia Learning* (pp. 17-133). New York: Cambridge University Press.

Flynt, J.P., & Vinson, B. (2005). *Simulation and Event Modeling for Game Developers*. Boston, MA: Thomson Course Technology.

Gaffney, C., Dagger, D., & Wade, V. (2008). A Survey of Soft Skill Simulation Authoring Tools. *Proceedings of the nineteenth ACM Conference on Hypertext and Hypermedia*, 181-185.

Ganapathy, S., Narayanan, S., & Srinivasan, K. (2003). Simulation Based Decision Support for Supply Chain Logisitics. *Proceedings of the 2003 Winter Simulation Conference*, 1013-1020.

Garrido, J.M. (1999). *Practical Process Simulation Using Object-Oriented Technique and C++*. Boston: Artech House.

Garrido, J.M. (2001). *Object-Oriented Discrete-event Simulation: A Practical Introduction*. New York: Kluwer Academic/Plenum Publishers.

Garrot, T., Psillaki, M., & Rochhia, S. (2008). Describing E-learning Development in European Higher Education Institutions Using a Balanced Scorecard. *The Economics of E-learning, 5*(1), 57-71.

Gelenbe, E., & Pujolle, G. (1998). *Introduction to Queing Network*. New York: Wiley.

George, D., & Mallery, P. (2009). *SPSS for Windows Step by Step: A Simple Guide and Reference 18.0 Update*. Boston: Pearson Allyn and Bacon.

Getting Started with SIMPROCESS. (2006). Retrieved September 6, 2008, from http://www.renque.com/downloads/RenqueManual.pdf

Geuder, D.F. (1995). Object Oriented Modeling with Simple++. *Proceedings of the 1995 Winter Simulation Conference*, 534-540.

Gibson, D., Aldrich, C., & Prensky, M. (2007). *Games and Simulations in Online Learning: Research and Development Frameworks*. Hershey, PA: Information Science Publishing.

Gilman, A. (1985). Interactive Control of the Model: A Natural Companion to Animated Simulation Graphics. *Proceedings of the 1985 Winter Simulation Conference*, 196-198.

Goble, J. (1991). Introduction to SIMFACTORY II.5. *Proceedings of the 1991 Winter Simulation Conference*, 77-80.

Goble, J. (1997). MODSIM III - A Tutorial. *Proceedings of the 1997 Winter Simulation Conference*, 601-605.

Gokhale, A.A. (1996). Effectiveness of Computer Simulation for Enhancing Higher Order Thinking. *Journal of Industrial Teacher Education, 33*(4), 36-46.

Goldman, S.R. (2003). Learning in Complex Domains: When and Why Do Multiple Representations Help? *Learning and Instruction, 13*(2), 239-244.

Gonzalez-Barbone, V., & Anido-Rifon, L. (2010). From SCORM to Common Cartridge: A step forward. *Computers & Education, 54*(1), 88-102.

Gredler, M.E. (2003). Games and Simulations and Their Relationships to Learning. In D. Jonassen (Ed.), *Handbook of Research for Educational Communications and Technology* (2nd ed., pp. 571-581). Mahwah, NJ: Lawrence Erlbaum Associates.

Greenbaum, J., & Kyng, M. (1991). *Design at Work: Cooperative Design of Computer Systems*. New Jersey: Lawrence Erlbaum Associates.

Haapala, A. (2006). Promoting Different Kinds of Learners towards Active Learning in the Web-Based Environment. *Informatics in Education, 2*(2), 207-218.

Hailikari, T., Katajavuori, N., & Lindblom-Ylanne, S. (2008). The Relevance of Prior Knowledge in Learning and Instructional Design. *American Journal of Pharmaceutical Education, 72*(5).

Halpin, B. (1999). Simulation in Sociology. *American Behaviroral Scientist, 42*(10), 1488-1508.

Hamlin, J.S., Tarbell, J., & Williams, B. (2003). *The Hidden Power of Flash Components*. San Francisco: Sybex.

Hannon, B., Ruth, M., & Meadows, D.H. (2001). *Dynamic Modeling* (2nd ed.). New York: Springer.

Harrel, C.R., & Price, R.N. (2003). Simulation Modeling Using ProModel Technology. *Proceedings of the 2003 Winter Simulation Conference*, 175-181.

Harrell, C., Ghosh, B.K., & Bowden, R.O. (2004). *Simulation Using ProModel* (2nd ed.). New York: McGraw Hill.

Hart, S.G., Stavenland, L.E., Hancock, P.A., & Meshkati, N. (1988). Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In P. A. Hancock & N. Meshkati (Eds.), *Human Mental Workload* (pp. 139-183). Armsterdan: Elsevier Science Publisher.

Healy, K.J., & Kilgore, R.A. (1998). Introduction to SILK and Java-based Simulation. *Proceedings of the 30th Conference on Winter Simulation*, 327-334.

Hegarty, M. (2004). Dynamic Visualizations and Learning: Getting to the Difficult Questions. *Learning and Instruction, 14*, 343–351

Hegarty, M., Kriz, S., & Cate, C. (2003). The Roles of Mental Animations and External Animations in Understanding Mechanical Systems. *Cognition and Instruction, 21*(4), 325-360.

Heinich, R., Molenda, M., Russell, J.D., & Smaldino, S.E. (1999). *Instructional Media and Technologies for Learning* (6 ed.). Upper Saddle River, N.J: Merrill.

Henderson, R., & Divett, M.J. (2003). Perceived Usefulness, Ease of Use and Electronic Supermarket Use. *International Journal of Human-Computer Studies, 59*(3), 383-395.

Henriksen, J.O. (1997). An Introduction to SLX. *Proceedings of the 1997 Winter Simulation Conference*, 559-566.

Henriksen, J.O. (2000). Adding Animation to a Simulation Using PROOF. *Proceedings of the 2000 Winter Simulation Conference*, 191-196.

Herrington, J., & Oliver, R. (1995). Critical Characteristics of Situated Learning: Implications for the Instructional Design of Multimedia. *Proceedings of ASCILITE'95*.

Herrington, J., & Oliver, R. (1997). Multimedia, Magic and the Way Students Respond to a Situated Learning Environment. *Australian Journal of Educational Technology, 13*(2), 127-143.

Hill, D.R.C. (1996). *Object-Oriented Analysis and Simulation*. Harlow, New York: Addison-Wesley.

Holzinger, A., & Ebner, M. (2003). Interaction and Usability of Simulations & Animations: A Case Study of the Flash Technology. *Proceedings of International Conference on Human-Computer Interactions 2003 (INTERACT'03)*, 777-780.

Hoppensteadt, F.C., & Peskin, C.S. (2002). *Modelling and Simulation in Medicine and Life Science*. New York: Springer.

Huk, T. (2006). Who Benefits from Learning with 3D Models? The Case of Spatial Ability. *Journal of Computer Assisted Learning, 22*(6), 392-404. doi: 10.1111/j.1365-2729.2006.00180.x

Hull, T.E., & Dobell, A.R. (1962). Random Number Generators. *SIAM Review, 4*(3), 230-254.

Hunter, D., Cagle, K., Gibbons, D., Ozu, N., Pinnock, J., & Spencer, P. (2000). *Beginning XML*. Birmingham: Wrox.

Iazeolla, G., & Ambrogio, A.D. (1998). Distributed Systems for Web-based Simulation. *Advances in Computer and Information Science'98*, 1-8.

Idrus, H., Dahan, H.M., & Abdullah, N. (2009). Challenges in the Integration of Soft Skills in Teaching Technical Courses: Lecturers' Perspectives. *Asian Journal of University Education, 5*(2), 67-81.

Igbaria, M., Livari, J., & Maragahh, H. (1995). Why Do Individuals Use Computer Technology?: A Finnish Case Study. *Information & Management, 29*(5), 227-238. doi: 10.1016/0378-7206(95)00031-0

Igbaria, M., Zinatelli, N., Cragg, P., & Cavaye, A. (1997). Personal Computing Acceptance Factors in Small Firms: A Structural Equation Model. *MIS Quarterly*(279-302).

Illeris, K. (2000). *The Three Dimensional of Learning: Contemporary Learning Theory in the Tension Field between the Cognitive, the Emotional and the Social*. Frederiksberg: Roskilde University Press.

Isomaki, H., Pekkola, S., & Bannon, L.J. (2011). "20 Years a-Growing": Revisiting From Human Factors to Human Actors *Reframing Humans in Information Systems Development* (Vol. 201, pp. 181-188). London: Springer

Jacobs, P.H.M., Lang, A.N., & Verbraeck, A. (2002). D-SOL; A Distributed Java Based Discrete Event Simulation Architecture. *Proceedings of the 2002 Winter Simulation Conference*, 793-800.

Jahangir, N., & Begum, N. (2008). The Role of Perceived Usefulness, Perceived Ease of Use, Security and Privacy, and Customer Attitude to Engender Customer

Adaptation in the Context of Electronic Banking. *African Journal of Business Management, 2* (1), 32-40.

Jeffries, P.R. (2005). A Framework for Designing, Implementing, and Evaluating: Simulations Used as Teaching Strategies in Nursing. *Nursing Education Perspectives, 26*(2), 96-103.

Jifeng, H., Li, X., & Liu, Z. (2005). Component-Based Software Engineering - the Need to Link Methods and their Theories. *Lecture Notes in Computer Science, 3722*, 70-95.

Johnson, S.D., Aragon, S.R., Shaik, N., & Palma-Rivas, N. (2000). Comparative Analysis of Learner Satisfaction and Learning Outcomes in Online and Face-to-face Learning Environments. *Journal of Interactive Learning Research, 11*(1), 29-49.

Jonassen, D.H., & Land, S.M. (2000). *Theoritical Foundations of Learning Environment*. New Jersey: Lawrence Erlbaum Associates.

Jong, T.D. (1991). Learning and Instruction with Computer Simulations. *Education & Computing, 6*(3-4), 217-229

Jong, T.D., & Joolingen, W.R.V. (1998). Scientific Discovery Learning with Computer Simulations of Conceptual Domains. *Review of Educational Research, 68*(2), 179-201.

Jong, T.D., & Joolingen, W.R.V. (2008). Model-Facilitated Learning. In J. M. Spector, M. D. Merrill, J. v. Merrienboer & M. P. Driscoll (Eds.), *Handbook of Research on Educational Communications and Technology* (pp. 457-468). New York: Taylor & Francis Group.

Jong, T.D., Martin, E., Zamarro, J.M., Esquembre, F., Swaak, J., & Joolingen, W.R.V. (1999). The Integration of Computer Simulation and Learning Support: An Example from the Physics Domain of Collisions. *Journal of Research in Science Teaching, 36*(5), 597-615.

Joolingen, W.R.V., & Jong, T.D. (1991a). Characteristics of Simulations for Instructional Settings. *Education & Computing, 6*(3-4), 241-262.

Joolingen, W.R.V., & Jong, T.D. (1991b). Supporting Hypothesis Generation by Learners Exploring an Interactive Computer Simulation. *Instructional Science, 20*(5), 389-404.

Kacer, J. (2002). Discrete Event Simulations with J-Sim. *Proceedings of the Inaugural Conference on the Principles and Practice of Programming*, 13-18.

Kalra, D., & Barr, A.H. (1992). Modeling with Time and Events in Computer Simulations. *Eurographics '92*, 45-58.

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. (2003). The Expertise Reversal Effect. *Educational Psychologist 38*(1), 23-31.

Kamat, V.R., & Martinez, J.C. (2001). Enabling Smooth and Scaleable Dynamic 3D Visualization of Discrete-Event Construction Simulations. *Proceedings of the 2001 Winter Simulation Conference*, 1528-1533.

Kamat, V.R., & Martinez, J.C. (2007). Variable-Speed Resource Motion in Animations of Discrete-Event Process Models. *Electronic Journal of Information Technology in Construction (ITcon), 12*, 293-303.

Kauchak, D.P., & Eggen, P.D. (2007). *Learning and Teaching: Research Based Methods*. Boston: Pearson Allyn & Bacon.

Kaye, J., & Castillo, D. (2003). *Flash MX for Interactive Simulation*. New York: Thompson Delmar Learning.

Kazymyr, V., & Demshevska, N. (2001). Application of Java-technologies for Simulation in the Web. *Proceedings of the 2001 International Conference on Information Systems Technology and Its Applications*, 173-184.

Keen, R.E., & Spain, J.D. (1992). *Computer Simulation in Biology*. New York: Wiley-Liss.

Keil, M., Beranek, P.M., & Konsynski, B.R. (1995). Usefulness and Ease of Use: Field Study Evidence Regarding Task Considerations. *Decision Support Systems 13*, 75-91.

Kelton, W.D., Sadowski, R.P., & Sturrock, D.T. (2004). *Simulation with Arena* (3rd ed.). New York: Mc-Graw Hill.

Kelton, W.D., Sadowski, R.P., & Swets, N.B. (2010). *Simulation with Arena* (5th ed.). Singapore: Mc Graw Hill.

Kennepohl, D. (2001). Using Computer Simulations to Supplement Teaching Laboratories in Chemistry for Distance Delivery. *Journal of Distance Education, 16*(2), 58-65.

Khalid, R., Kreutzer, W., & Bell, T. (2009). Combining Simulation and Animation of Queueing Scenarios in a Flash-based Discrete Event Simulator. *Lecture Notes in Business Information Processing, 20*, 240-251.

Kilgore, R.A. (2000). Silk, Java and Object-Oriented Simulation. *Proceedings of the 2000 Winter Simulation Conference*, 246-252.

Kim, J.O., & Mueller, C.W. (1978). *Introduction to Factor Analysis: What It Is and How To Do it*. Newbury Park: Sage Publications.

Kim, K. (2006). The Future of Online Teaching and Learning in Higher Education: The Survey Says. *EDUCAUSE Quarterly, 29*(4), 22-30.

Kirschner, P.A., Sweller, J., & Clark, R.E. (2006). Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching. *Educational Psychologist, 41*(2), 75-86.

Klein, U., Straßburger, S., & Beikirch, J. (1998). Distributed Simulation with JavaGPSS Based on the High Level Architecture. *Proceedings of the 1998 SCS International Conference on Web-Based Modeling and Simulation*, 85-90.

Klobas, J., & McGill, T. (2010). The Role of Involvement in Learning Management System Success. *Journal of Computing in Higher Education, 22*(2), 114-134. doi: 10.1007/s12528-010-9032-5

Knowles, M.S. (1984). *Andragogy in Action*. San Francisco: Jossey-Bass.

Knuth, D.E. (1981). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (2nd ed.). Reading: Addison-Wesley.

Kolb, D.A. (1984). *Experiental Learning: Experience as the Source of Learning and Development*. Englewood Cliffs: NJ: Prentice-Hall.

Korakakis, G., Pavlatou, E.A., Palyvos, J.A., & Spyrellis, N. (2009). 3D Visualization Types in Multimedia Applications for Science Learning: A Case Study for 8th Grade Students in Greece. *Computers & Education, 52*(2), 390-401.

Kozma, R. (2003). The Material Features of Multiple Representations and Their Cognitive and Social Affordances for Science Understanding. *Learning and Instruction, 13*(2), 205-226.

Krahl, D. (2003). Extend: An Interactive Simulation Tool. *Proceedings of the 2003 Winter Simulation Conference*, 188-196.

Krahl, D. (2007). *ExtendSim7. Proceedings of the 2007 Winter Simulation Conference*, 226-232.

Krathwohl, D.R., Bloom, B.S., & Masia, B.B. (1996). *Taxonomy of Educational Objectives, Handbook 1: Affective Domain* (2nd ed.). New York: Longman.

Kreiman, J., & Mullarney, A. (1987). *SIMSCRIPT II.5 Programming Language* (4th ed.). Los Angeles, CA: CACI.

Kreutzer, W. (1986). *System Simulation: Programming Styles and Languages*. Boston: Addison-Wesley Publisher Limited.

Kreutzer, W., Hopkins, J., & Mierlo, M.C. (1997). *SimJAVA: A Framework for Modelling Queing Networks in Java*. Paper presented at the Proceedings of the 1997 Winter Simulation Conference, Atlanta, GA.

Kreutzer, W., Hopkins, J., & Mierlo, M.V. (1997). SimJAVA - A Framework for Modeling Queueing Networks in Java. *Proceedings of the 29th Conference on Winter Simulation*, 483-488. doi: http://doi.acm.org/10.1145/268437.268548

Kühl, T., Scheiter, K., Gerjets, P., & Gemballa, S. (2011). Can Differences in Learning Strategies Explain the Benefits of Learning from Static and Dynamic Visualizations?. *Computers & Education, 56*(1), 176-187.

Kuljis, J., & Paul, R.J. (2000). A Review of Web Based Simulation: Whither We Wander?. *Proceedings of the 2000 Conference on Winter Simulation*, 1872-1881.

L'Ecuyer, P., Meliani, L., & Vaucher, J. (2002). SSJ: A Framework for Stochastic Simulation in Java. *Proceedings of the 2002 Winter Simulation Conference*, 234-242.

Laitenberger, O., & Dreyer, H.M. (1998). Evaluating the Usefulness and the Ease of Use of a Web-based Inspection Data Collection Tool. *Proceedings of Fifth International on Software Metrics Symposium, 1998 (Metrics 1998)*, 122-132.

Lambert, K.A., & Osborne, M. (2004). *Java: A Framework for Program Design and Data Structures*. Belmont, CA: Thomson-Brooks/Cole.

Land, S. (2000). Cognitive Requirements for Learning with Open-ended Learning Environments. *Educational Technology Research and Development, 48*(3), 61-78.

Landriscina, F. (2009). Simulation and Learning: The Role of Mental Models. *Journal of e-Learning and Knowledge Society, 5*(2), 23-32.

Lau, Y.-T. (2000). *The Art of Objects: Object-Oriented Design and Architecture*. Upper Saddle River: Addison-Wesley Professional

Law, A.M. (2007). *Simulation Modeling and Analysis* (4 ed.). Boston: McGraw-Hill.

Law, A.M., & Kelton, W.D. (2000). *Simulation Modeling and Analysis*. New York: McGraw-Hill.

LeBaron, T., & Jacobson, C. (2007). The Simulation Power of AutoMOD. *Proceedings of the 2007 Winter Simulation Conference*, 210-218.

Ledin, J. (2001). *Simulation Engineering: Build Better Embedded Systems Faster*. Lawrence, KS: CMP Books.

Lee, J. (1999). Effectiveness of Computer-Based Instructional Simulation: A Meta Analysis. *International Journal of Instructional Media, 26*(1), 71-85.

Legris, P., Ingham, J., & Collerette, P. (2003). Why Do People Use Information Technology? A Critical Review of the Technology Acceptance Model. *Information & Management, 40*(3), 191-204.

Leutner, D. (1993). Guided Discovery Learning with Computer-based Simulation Games: Effects of Adaptive and Non-adaptive Instructional Support. *Learning and Instruction, 3*(2), 113-132.

Liao, T.T., & Miller, D.C. (1996). Computer Games: Increase Learning in an Interactive Multidisplinary Environment. *Journal of Educational Technology Systems, 24*(2), 195-205.

Little, M.C., & McCue, D.L. (1993). Construction and Use of a Simulation Package in C++: University of Newcastle Upon Tyne.

Livesey, P.J. (1986). *Learning and Emotion: A Biological Synthesis*. Hillsdale, N.J.: Lawrence Erlbaum Associates.

Lomow, G., & Baezner, D. (1989). A Tutorial Introduction to Object-Oriented Simulation and SIM++. *Proceedings of the 1989 Winter Simulation Conference*, 140-146.

Lopez, L.A. (2006). *New Perspective on Macromedia Flash 8: Comprehensive*. Boston: Thompson Course Technology.

Lott, J., & Patterson, D. (2007). *Advanced ActionScript 3 with Design Patterns*. Berkeley, CA: Peachpit Press.

Lowe, R. (2004). Interrogation of a Dynamic Visualization During Learning. *Learning and Instruction, 14*(3), 257-274.

Lunce, L.M. (2004). Computer Simulations in Distance Education. *International Journal of Instructional Technology and Distance Learning, 1*(10), 29-40.

Lunce, L.M. (2006). Simulations: Bringing the Benefits of Situated Learning to the Traditional Classroom. *Journal of Applied Educational Technology, 3*(1), 37-45.

m-Plant: Empower for Manufacturing Process Management. (2003). from http://www.sim-serv.com/pdf/tools/tool_14.pdf

Macal, C.M. (2001). Simulation and Visualization. *SIMULATION, 77*(49), 90-92.

Maldonado, H., Lee, J.-E.R., Brave, S., Nass, C., Nakajima, H., Yamada, R. (2005). We Learn Better Together: Enhancing eLearning with Emotional Characters. *Proceedings of the 2005 Conference on Computer Support for Collaborative Learning 2005: The Next 10 Years*, 408-417.

Markowitz, H., Hausner, B., & Karr, H.W. (1963). *SIMSCRIPT: A Simulation Programming Language*. Englewood Cliffs, NJ: Prentice-Hall.

Martinez, M. (2000). International Learning in an International World. *ACM Journal of Computer Documentation, 24*(1), 3-20. doi: http://doi.acm.org/10.1145/330409.330411

Mascarenhas, E., Rego, V., & Sang, J. (1995). DISplay: A System for Visual-Interaction in Distributed Simulations. *Proceedings of the 1995 Winter Simulation Conference*, 698-705.

Mathieson, K. (1991). Predicting User Intentions: Comparing the Technology Acceptance Model with the Theory of Planned Behavior. *Information Systems Research, 2*(3), 173-191.

Matloff, N. (2008). Introduction to Discrete-Event Simulation and the SimPy Language. Retrieved September 2008, 2008, from http://heather.cs.ucdavis.edu/~matloff/156/PLN/DESimIntro.pdf

Matwiczak, K.M. (1990). Interactive Simulation: Let the User Beware. *Proceedings of the 1990 Winter Simulation Conference*, 453-456.

Mayer, R.E. (2003). Elements of a Science of E-learning. *Journal of Educational Computing Research 29*(3), 297 - 313

Mayer, R.E., Hegarty, M., Mayer, S., & Campbell, J. (2005). When Static Media Promote Active Learning: Annotated Illustrations Versus Narrated Animations in Multimedia Instruction. *Journal of Experimental Psychology: Applied, 11*(4), 256–265.

Mayer, R.E., & Moreno, R. (2003). Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational Psychologist, 38*(1), 43-52.

McKenna, P., & Laycock, B. (2004). Constructivist or Instructivists Pedagogical Concepts Practically Applied to a Computer Learning Environment. *ACM SIGCSE Bulletin, 36*(3), 166-170.

McNab, R., & Howell, F.W. (1996). Using Java for Discrete Event Simulation. *Proceeding of Twelfth UK Computer and Telecommunications Performance Engineering Workshop (UKPEW)*, 219-228.

McNab, R., & Howell, F.W. (1998). simjava: A Discrete Event Simulation Package for Java with Applications in Computer Systems Modelling. *Proceeding of the First International Conference on Web-based Modelling and Simulation*.

Melão, N., & Pidd, M. (2007). Using Component Technology to Develop a Simulation Library for Business Process Modelling. *European Journal of Operational Research, 172*(1), 163-178.

Meyer, R., Page, B., Kreutzer, W., Knaak, N., & Lechler, T. (2005a). DESMO-J - A Framework for Discrete Event Modelling & Simulation. In B. Page & W. Kreutzer (Eds.), *The Java Simulation Handbook - Simulating Discrete Event Systems with UML and Java* (pp. 263-335). Aachen: Shaker Verlag.

Meyer, R., Page, B., Kreutzer, W., Knaak, N., & Lechler, T. (2005b). DESMO-J - A Framework for Discrete Event Modelling & Simulation. In B. Page & W. Kreutzer (Eds.), *Simulating Discrete Event Systems with UML and Java*. Aachen: Shaker Verlag.

Michael, K.Y. (2000). *A Comparison of Students' Product Creativity Using a Computer Simulation Activity Versus a Hands-on Activity in Technology Education.* Virginia Polytechnic Institute and State University.

Michelson, J.D., & Manning, L. (2008). Competency Assessment in Simulation-based Procedural Education. *The American Journal of Surgery, 196*(4), 609-615.

Mildrad, M. (2002). Using Construction Kits, Modeling Tools and System Dynamics Simulations to Support Collaborative Discovery Learning. *Educational Technology & Society, 5*(4), 76-87.

Miller, J.A., Ge, Y., & Tao, J. (1998). Component-Based Simulation Environment: JSIM as a Case Study Using Java Beans. *Proceedings of the 1998 Winter Simulation Conference*, 373-381.

Miller, J.A., Ge, Y., & Tao, J. (1998). *Component-based Simulation Environments: JSIM as a Case Study Using Java Beans*. Paper presented at the Proceedings of the 30th conference on Winter simulation, Washington, D.C., United States.

Milrad, M. (2002). Using Construction Kits, Modeling Tools and System Dynamics Simulations to Support Collaborative Discovery Learning. *Educational Technology & Society, 5*(4), 76-87.

Min, R. (2003). Simulation and Discovery Learning in an Age of Zapping and Searching: Learning Models. *Turkish Online Journal of Distance Education, 4*(2).

Mohler, J.L. (2006). *Flash 8: Graphics, Animation and Interactivity*. New York: Thomson/Delmar Learning.

Moock, C. (2002). *ActionScript for Flash MX: The Definitive Guide, Second Edition* (2 ed.). Sebastopol: O'Reilly Media.

Moock, C. (2004). *Essential ActionScript 2.0*. Farnham: O'Reilley.

Moreno, R. (2006). Does the Modality Principle Hold for Different Media? A Test of the Method-Affects-Learning Hypothesis. *Journal of Computer Assisted Learning, 22*(3), 149-158. doi: 10.1111/j.1365-2729.2006.00170.x

Moreno, R., & Mayer, R. (2007). Interactive Multimodal Learning Environments. *Educational Psychology Review, 19*(3), 309-326. doi: 10.1007/s10648-007-9047-2

Moretti, S. (2002). Computer Simulation in Sociology: What Contributions?. *Social Science Computer Review, 20*(1), 43-57.

Narayanan, N.H., & Hegarty, M. (2002). Multimedia Design for Communication of Dynamic Information. *International Journal of Human Computer Studies, 57*(4), 279-315. doi: http://dx.doi.org/10.1006/ijhc.2002.1019

Narayanan, S., Cowgill, J., Malu, P., Nandha, H., Patel, C., Schneider, N. (1997). Web-based Distributed Interactive Simulation Using Java. *Proceedings of the 1997 IEEE International Conference on Systems, Manufacturing and Cybernetics, 3*, 2690-2695.

Neumann, G., Page, B., Kreutzer, W., Kiesel, G., & Meyer, R. (2005). Simulation and E-Learning. In B. Page & W. Kreutzer (Eds.), *Simulating Discrete Event Systems with UML and Java* (pp. 401-433). Aachen: Shaker Verlag.

Nigel, N. (2008). *Curriculum and the Teacher: 35 years of the Cambridge Journal of Education*. London: Routledge.

Njoo, M., & Jong, T.D. (1993). Exploratory Learning with a Computer Simulation for Control Theory: Learning Processes and Instructional Support. *Journal of Research in Science Teaching, 30*(8), 821-844.

Noguez, J., & Sucar, L. (2005). A Semi-open Learning Environment for Virtual Laboratories *MICAI 2005: Advances in Artificial Intelligence* (pp. 1185-1194).

Nordgren, W.B. (2003). Flexsim Simulation Environment. *Proceedings of the 2003 Winter Simulation Conference*, 197-200.

O'Reilly, J. (2002). Introduction to AweSim. *Proceedings of the 2002 Winter Simulation Conference*, 221-224.

Odhabi, H.I., Paul, R.J., & Macredie, R.D. (1998). Developing a Graphical User Interface for Discrete Event Simulation. *Proceedings of the 1998 Winter Simulation Conference*, 429-436.

Oloruntegbe, K.O., & Alam, G.M. (2010). Evaluation of 3d Environments and Virtual Realities in Science Teaching and Learning: The Need to Go Beyond Perception Referents. *Scientific Research and Essays, 5*(9), 948-954.

Oses, N., Pidd, M., & Brooks, R.J. (2004). Critical Issues in the Development of Component-based Discrete Simulation. *Simulation Modelling Practice and Theory, 12*(7-8), 495-514.

Paas, F., Tuovinen, J., Tabbers, H., & Gerven, P.V. (2003). Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. *Educational Psychologist, 38*(1), 63-71.

Page, B., & Kreutzer, W. (2005). *The Java Simulation Handbook: Simulating Discrete Event Systems with UML and Java*. Aachen: Shaker Verlag.

Page, E.H., Moose, R.L.J., & P.Griffin, S. (1997). Web-Based Simulation in Simjava Using Remote Method Invocation. *Proceedings of the 1997 Winter Simulation Conference*, 468-473.

Parrish, P. (2009). Aesthetic Principles for Instructional Design. *Educational Technology Research and Development, 57*(4), 511-528. doi: 10.1007/s11423-007-9060-7

Payne, J.W. (1982). Contingent Decision Behavior. *Psychological Bulletin, 92*(2), 382-402.

Pedgen, C.D. (2007). Simio: A New Simulation System Based on Intelligent Objects. *Proceedings of the 2007 Winter Simulation Conference*, 2293-2300.

Pegden, C.D. (1989). *Introduction to SIMAN.* Sewickley, PA: Systems Modelling Cooperation.

Pegden, C.D., Alan, A., & Pritsker, B. (1978). SLAM Tutorial. *Proceedings of the 1982 Winter Simulation Conference*, 661-668

Pegden, C.D., Shannon, R.E., & Sadowski, R.P. (1995). *Introduction to Simulation Using Siman* (2nd ed.). New York: McGraw-Hill.

Peters, K., & Yard, T. (2004). *Extending Macromedia Flash MX 2004: Complete Guide and Reference to JavaScript Flash.* Birmingham: Friends of ED.

Piaget, J. (1977). *The Development of Thought: Equilibration of Cognitive Structures.* Oxford: B. Blackwell.

Pidd, M. (2004). *Computer Simulation in Management Sciences* (5th ed.). Hoboken, NJ: Wiley.

Pikkarainen, T., Pikkarainen, K., Karjaluoto, H., & Pahnila, S. (2004). Consumer Acceptance of Online Banking: An Extension of the Technology Acceptance Model. *Internet Research, 14*(3), 224-235.

Pilkington, R., & Parker-Jones, C. (1996). Interacting with Computer-based Simulation: The Role of Dialogue. *Computers and Education, 27*(1), 1-14.

Porter, T.S., Riley, T.M., & Ruffer, R.L. (2004). A Review of the Use of Simulations in Teaching Economics. *Social Science Computer Review, 22*(4), 426-443.

Praehofer, H., Sametinger, J., & Stritzinger, A. (2001). Concepts and Architecture of a Simulation Framework Based on the JavaBeans Component Model. *Future Generation Computer Systems, 17*(5), 539-559.

Prensky, M. (2001). *Digital Game-Based Learning.* New York: McGraw-Hill.

Pritsker, A.A.B., & O'Reilly, J.J. (1999). *Simulation with Visual SLAM and AweSim.* New York: John Wiley & Sons.

Pritsker, A.A.B., Sigal, C.E., & Hammesfahr, R.D.J. (1994). *SLAM II: Network Models for Decision Support.* New York: Scientific Press.

Quinn, C.N. (2005). *Engaging Learning: Designing e-Learning Simulation Games.* San Francisco: Pfeiffer.

Quinn, J., & Alessi, S. (1994). The Effects of Simulation Complexity and Hypothesis-generation Strategy on Learning. *Journal of Research on Computing in Education 27*(1), 75-91.

Radcliff, J.B. (2005). Why Soft Skill Simulation. www.competenet.com/downloads/SimulationWP-F1.pdf

Reid, D.J., Zhang, J., & Chen, Q. (2003). Supporting Scientific Discovery Learning in a Simulation Environment. *Journal of Computer Assisted Learning, 19*, 9-20.

Rekapalli, P.V., & Martinez, J.C. (2007). A Message-Based Architecture to Enable Runtime User Interaction on Concurrent Simulation-Animations of Construction Operations. *Proceedings of the 2007 Winter Simulation Conference*, 2028-2031

Renque Discrete Event Simulation: User's Guide. (2008). Retrieved September, 6, 2008, from http://www.renque.com/downloads/RenqueManual.pdf

Renshaw, C.E., & Taylor, H.A. (2000). The Educational Effective of Computer-based Instruction. *Computer & Geocities, 26,* 677-682.

Repenning, A., Ioannidou, A., Payton, M., Ye, W., & Roschelle, J. (2001). Using Components for Rapid Distributed Software Development. *Journal of Software, 18*(2), 38-45.

Rice, S.V., Marjanski, A., M., M.H., & Bailey, S.M. (2004). Object Oriented SIMSCRIPT. *Proceedings of the 37th Annual Simulation Symposium*, 178-187.

Rice, S.V., Marjanski, A., Markowitz, H.M., & Bailey, S.M. (2005). The SIMSCRIPT III Programming Language for Modular Object-Oriented Simulation. *Proceedings of 2005 Winter Simulation Conference*, 621-630.

Rieber, L.P. (1992). Computer-based Microworlds: A bridge between Constructivism and Direct Instruction. *Educational Technology Research and Development, 40*(1), 93-106.

Rieber, L.P. (1995). Using Computer-based Microworlds with Children with Pervasive Developmental Disorders: An Informal Case Study. *Journal of Educational Multimedia and Hypermedia, 4*(1), 75-94.

Rieber, L.P. (1996). Seriously Considering Play: Designing Interactive Learning Environments Based on the Blending of Microworlds, Simulations, and Games. *Educational Technology Research & Development, 44*(2), 43-58.

Rieber, L.P. (2002). Supporting Discovery-based Learning with Simulations. *The International Workshop on Dynamic Visualizations and Learning, Knowledge Media Research Center.*

Rieber, L.P., Tzeng, S.-C., & Tribble, K. (2004). Discovery learning, representation, and explanation within a computer-based simulation: finding the right mix. *Learning and Instruction, 14*(3), 307-323.

River, R.H., & Vockell, E. (1987). Computer Simulations to Stimulate Scientific Problem Solving. *Journal of Research in Science Teaching, 24*, 403-415.

Rob, P., & Semaan, E. (2000). *Databases: Design, Development and Deployment.* Singapore: McGraw-Hill Higher Education.

Robinson, S.L. (1994). An Introduction to Visual Interactive Simulation in Business. *International Journal of Information Management, 14*(1), 13-23.

Robinson, W.R. (2000). A View of the Science Education Research Literature: Scientific Discovery Learning with Computer Simulations. *Journal of Chemical Education, 77*(1), 17. doi: 10.1021/ed077p17

Rohrer, M.W. (2000). Seeing is Believing: The Importance of Visualization in Manufacturing Simulation. *Proceedings of the 2000 Winter Simulation Conference*, 1211-1216.

Romiszowski, A. (2004). How's the E-learning Baby? Factors Leading to Success or Failure of an Educational Technology Innovation. *Educational Technology, 44*(1), 5-27.

Rooks, M. (1991). A Unified Framework for Visual Interactive Simulation. *Proceedings of the 1991 Winter Simulation Conference*, 1146-1155.

Roschelle, J., DiGiano, C., Koutlis, M., Repenning, A., Phillips, J., Jackiw, N. (1999). Developing Educational Software Components. *Journal of Computer, 32*(9), 50 - 58

Rose, L.L. (1981). Hierarchical Modelling in GASP. *Proceedings of the 14th Annual Symposium on Simulation*, 199-213.

Rossetti, M.D., Aylor, B., Jacoby, R., Prorock, A., & White, A. (2000). SIMFONE: An Object-Oriented Simulation Framework. *Proceedings of the 2000 Winter Simulation Conference*, 1855-1864.

Rosson, M.B., & Seals, C.D. (2001). Teachers as Simulation Programmers: Minimalist Learning and Reuse. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 237-244.

Saadé, R., & Bahli, B. (2005). The Impact of Cognitive Absorption on Perceived Usefulness and Perceived Ease of Use in On-line Learning: An Extension of the Technology Acceptance Model. *Information & Management, 42*(2), 317-327.

Sahin, S. (2006). Computer Simulations in Science Education: Implications for Distance Education. *Turkish Online Journal of Distance Education, 7*(4), 132-146.

Sanders, W.B. (2004). *Macromedia Flash MX Professional 2004: Kick Start.* Indianapolis: Sams.

Sanders, W.B., & Cumaranatunge, C. (2007). *ActionScript 3.0 Design Patterns.* Sebastapol, CA: O'Reilly.

Sargent, R.G. (2004). Some Recent Advances in the Process Worldview. *Proceedings of the 2004 Winter Simulation Conference*, 294-299.

Schank, R.C., Berman, T.R., & Macpherson, K.A. (1999). Learning by Doing. In C. M. Reigeluth (Ed.), *Instructional-Design Theories and Models: A New Paradigm of Instructional Theory, Vol. 2 (Instructional Design Theories & Models)*. Mahwah, NJ: Lawrence Erlbaum Associates.

Schnotz, W., & Bannert, M. (2003). Construction and Interference in Learning from Multiple Representation. *Learning and Instruction, 13* (2), 141-156.

Schwartz, D.L., Bransford, J.D., & Sears, D. (2005). Efficiency and Innovation in Transfer. In R. E. Haskell (Ed.), *Transfer of Learning from a Modern Multidisciplinary Perspective* (pp. 1-51). CT: Information Age Publishing.

Schwetman, H. (1988). Using CSIM to Model Complex Systems. *Proceedings of the 1988 Winter Simulation Conference*, 246-253.

Schwetman, H. (2001). CSIM19: A Powerful Tool for Building System Models. *Proceedings of the 2001 Winter Simulation Conference*, 250-255.

Seila, A.F. (1986). *Discrete Event Simulation in PASCAL with SIMTOOLS.* Paper presented at the Proceedings of the 18th Conference on Winter Simulation, Washington, DC.

Seufert, T. (2003). Supporting Coherence Formation in Learning from Multiple Representations. *Learning and Instruction, 13* (2), 227-237.

Shannon, R.E. (1998). Introduction to the Art and Science of Simulation. *Proceedings of the 30th Conference on Winter Simulation*, 7-14.

Sheppard, B.H., Hartwick, J., & Warshaw, P.R. (1998). The Theory of Reasoned Action: A Meta Analysis of Past Research with Recommendations for Modifications and Future Research. *Journal of Consumer Research, 15*(3), 325-343.

Shi, J.J., & Zhang, H. (1999). Iconic Animation of Construction Simulation. *Proceedings of the 1999 Winter Simulation Conference*, 992-997.

Shupe, R., & Hoekman, R. (2006). *Flash 8: Projects for Learning Animation and Interactivity*. Sebastopol: O'Reilley Media Inc.

Siemens, G. (2005). Connectivism: A Learning Theory for the Digital Age. *International Journal of Instructional Technology and Distance Learning, 2*(1), 3-10.

Sikora, A., & Niewiadomska-Szynkiewicz, E. (2007). A Federated Approach to Parallel and Distributed Simulation of Complex Systems. *International Journal of Applied Mathematics and Computer Sciences, 17*(1), 99–106.

Smialek, M. (2002). Developing e-Learning Simulations with Tools You Already Know. Retrieved May 12, 2008, from http://www.elearningguild.com/pdf/2/120302DEV-P.pdf

Smith, L.H., & Renzulli, J.S. (1984). Learning Style Preferences: A Practical Approach for Classroom Teachers. *Theory into Practice, 23*(1), 44-50.

Stahl, I. (2003). How Should We Teach Simulation. *Proceedings of the 2000 Winter Simulation Conference*, 1602-1612.

Stenalt, M.H., & Godsk, M. (2006). The Pleasure of E-Learning - Towards Aesthetic E-Learning Platforms. *Proceedings of the 12th International Conference of European University Information Systems*, 210-212.

Sterman, J.D. (2001). System Dynamics Modeling: Tools for Learning in a Complex World. *California Management Review, 43*(1), 8-25.

Stoel, L., & Lee, K.H. (2003). Modeling the Effect of Experience on Student Acceptance of Web-based Courseware. *Internet Research, 13* (5), 364 - 374.

Strassburger, S., Schulze, T., Lemessi, M., & Rehn, G.D. (2005). Temporally Parallel Coupling of Discrete Simulation Systems with Virtual Reality Systems. *Proceedings of the 2005 Winter Simulation Conference*, 1949-1957.

Su, B., Bonk, C.J., Magjuka, R.J., Liu, X., & Lee, S.-h. (2005). The Importance of Interaction in Web-Based Education: A Program-level Case Study of Online MBA Courses. *Journal of Interactive Online Learning, 4*(1), 1-18.

Swaak, J., & Jong, T.D. (2001a). Discovery Simulations and the Assessment of Intuitive Knowledge. *Journal of Computer Assisted Learning, 17*(3), 284-294.

Swaak, J., & Jong, T.D. (2001b). Learner vs. System Control in Using Online Support for Simulation-based Discovery Learning. *Learning Environments Research, 4*(3), 217-241.

Syrjakow, M., Berdux, J., & Szczerbicka, H. (2000). Interactive Web-based Animations for Teaching and Learning. *Proceedings of the 2000 Winter Simulation Conference*, 1651-1659.

Tan, J., & Biswas, G. (2007). Simulation-Based Game Learning Environments: Building and Sustaining a Fish Tank. *The First IEEE International Workshop on Digital Game and Intelligent Toy Enhanced Learning*, 73-80.

Taylor, S., & Todd, P.A. (1995). Understanding Information Technology Usage: A Test of Competing Models. *Information Systems Research, 6*(2), 144-176. doi: 10.1287/isre.6.2.144

Teo, T.S.H., Lim, V.K.G., & Lai, R.Y.C. (1999). Intrinsic and extrinsic motivation in Internet usage. *Omega, 27*(1), 25-37.

Teoh, B.S.-P., & Neo, T.-K. (2007). Using Computer-generated Animation as Additional Visual Elaboration in Undergraduate Courses. *The Turkish Online Journal of Educational Technology, 6*(4), 28-37.

Thomas, R.C., & Milligan, C.D. (2004). Putting Teachers in the Loop: Tools for Creating and Customizing Simulations. *Journal of Interactive Media in Education*(15).

Thompson, W.B. (1996). Introduction to the WITNESS Visual Interactive Simulator and OLEII Automation. . *Proceedings of the 1996 Winter Simulation Conference*, 547-550.

Tornatzky, L.G., & Klein, K.J. (1982). Innovation Characteristics and Innovation Adoption-Implementation: A Meta-Analysis of Findings. *IEEE Transactions on Engineering Management, 29*(1), 28-45.

Towne, D.M. (2007). *Enhancing Human Performance via Simulation-based Training and Aiding: A Guide to Design and Development*. Rotterdam: Sense Publishers.

Tumay, K. (1987). Factory Simulation with Animation: The No Programming Approach. *Proceedings of the 1987 Winter Simulation Conference*, 258-260.

Tversky, B., & Morrison, J. (2002). Animation: Can It facilitate?. *International Journal of Human-Computer Studies, 57*, 247-262.

Tyan, H.Y. (2002). *Design, Realization and Evaluation of a Component-Based Compositional Software Architecture for Network Simulation.* (PhD Thesis), The Ohio State University.

Valentine, E.C., Verbraeck, A., & Sol, H.G. (2003). Advantages and Disadvantages of Building Blocks in Simulation Studies: A Laboratory Experiment with Simulation Expert. *Proceedings of the 15th European Simulation Symposium*, 141-148.

Veeke, H.P.M., & Ottjes, J.A. (1999). Tomas: Tool for Object-Oriented Modelling and Simulation. *Proceedings of the Business and Industry Simulation Symposium*, 76-81.

Veermans, K., Jong, T.D., & Joolingen, W.R.V. (2000). Promoting Self-Directed Learning in Simulation-Based Discovery Learning Environments Through Intelligent Support. *Interactive Learning Environments, 8*(3), 229-255.

Venkatesh, V., & Davis, F.D. (2000). A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies. *Management Science, 46*(2), 186-204.

Venkatesh, V., & Morris, M. (2000). Why Don't Men Ever Stop to Ask for Directions? Gender, Social Influence, and Their Role in Technology Acceptance and Usage Behavior. *MIS Quarterly, 24*(1), 115-139.

Vogel-Walcutt, J.J., Gebrim, J.B., & Nicholson, D. (2010). Animated versus Static Images of Team Processes to Affect Knowledge Acquisition and Learning Efficiency. *Journal of Online Learning and Teaching, 6*(1), 163-173.

Vossen, G., & Westerkamp, P. (2006). Towards the Next Generation of E-Learning Standards: SCORM for Service-Oriented Environments. *Proceedings of Sixth International Conference on Advanced Learning Technologies*, 1031-1035.

Vries, J.D. (2004). Character-Based Simulations: What Works. http://www.openu.ac.il/research_center/download/CHARAC1.pdf

Wagner, E.D. (2006). Delivering on the Promise of eLearning. http://www.adobe.com/education/pdf/elearning/Promise_of_eLearning_wp_final.pdf

Wahlstedt, A., Pekkola, S., & Niemelä, M. (2008). From e-learning Space to e-learning Place. *British Journal of Educational Technology, 39*, 1020-1030. doi: 10.1111/j.1467-8535.2008.00821_1.x

Wainer, G.A., & Mosterman, P.J. (2010). *Discrete-Event Modeling and Simulation: Theory and Applications*. Boca Raton: CRC Press.

Warshaw, P.R., & Davis, F.D. (1985). Disentangling Behavioral Intention and Behavioral Expectation. *Journal of Experimental Social Psychology, 21*(3), 213-228.

Wenzel, S., & Jessen, U. (2001). The Integration of 3-D Visualization into the Simulation-based Planning Process of Logistics Systems. *SIMULATION, 77*(3-4), 114 -127.

White, B., Shimoda, T., & Frederiksen, J. (1999). Enabling Students to Construct Theories of Collaborative Inquiry and Reflective Learning: Computer Support for Metacognitive Development. *International Journal of AI in Education, 10*, 151-182.

Whiteside, J.A. (2002). Beyond Interactivity: Immersive Web-Based Learning Experiences. Retrieved May 12, 2007, from www.elearningguild.com/pdf/2/120302DEV-P.pdf

Whitworth, B., Banuls, V., Sylla, C., & Mahinda, E. (2008). Expanding the Criteria for Evaluating Socio-Technical Software. *IEEE Transactions on Systems, Manufacturing and Cybernetics, 38*(4), 777-790.

Wilson, B.G., Jonassen, D.H., & Cole, P. (1993). The ASTD Handbook of Instructional Technology. In G. M. Piskurich (Ed.), *Cognitive Approaches to Instructional Design* (pp. 21.21-21.22). New York: McGraw-Hill.

Wittrock, M.C. (1989). Generative Processes of Comprehension. *Educational Psychologist, 24*(4), 345.

Woo, Y., & Reeves, T. (2007). Meaningful Interaction in Web-based Learning: A Social Constructivist Interpretation. *Internet and Higher Education, 10*(1), 15-25.

Wright, P. (1998). *Beginning Visual Basic 6 Objects*. Indianapolis: Wrox Press.

Wurdinger, S.D., & Carlson, J. (2010). *Teaching for Experiential Learning: Five Approaches that Work*. Lanham: Rowman & Littlefield Education.

Yahiaoui, A., Hensen, J.L.M., & Soethout, L.L. (2004). Developing CORBA-based Distributed Control and Building Performance Environments by Run-time Coupling. *Proceedings of the 10th International Conference on Computing in Civil and Building Engineering*, 86-94.

Yi, M.R., & Cho, T.H. (2001). Hierarchical Simulation Model with Animation for Large Network Security. *Lecture Notes in Computer Science, 2229*, 456-460.

Yi, M.R., & Cho, T.H. (2003). Hierarchical Simulation Model with Animation. *Engineering with Computers, 19*(2), 203-212.

Yin, C., Ogata, H., & Yano, Y. (2007). Participatory Simulation Framework to Support Learning Computer Science. *International Journal of Mobile Learning and Organisation 1*(3), 288 - 304.

Zak, D. (2009). *Clearly Visual Basic programming with Microsoft Visual Basic 2008*. Boston: Course Technology.

Zeigler, B.P. (1984). *Multifaceted Modeling and Discrete Event Simulation*. London: Academic Press.

Zeigler, B.P. (1990). *Object Oriented Simulation with Modular, Hierarchical Models*. New York: Academic Press.

Zeigler, B.P. (2000). *Theory of Modeling and Simulation* (2nd ed.). San Diego: Academic Press.

Zhang, J., Chen, Q., Sun, Y., & Reid, D.J. (2004). Triple Scheme of Learning Support Design for Scientific Discovery Learning Based on Computer Simulation: Experimental Research. *Journal of Computer Assisted Learning, 20*, 269-282.

Zhong, Y., & Shirinzadeh, B. (2004). Analysis, Conversion and Visualization of Discrete Simulation Results. *Proceedings of the Eighth International Conference on Information Visualisation*, 118-123.