2011 International Conference on Computer and Software Modeling
IPCSIT vol.14 (2011) © (2011) IACSIT Press, Singapore

Component Based Modelling for Animated Educational Queuing Networks

Ruzelan Khalid[1], Wolfgang Kreutzer[2] and Tim Bell[2+]

[1]College of Arts and Sciences, Universiti Utara Malaysia, 06010 UUM Sintok, Kedah, Malaysia
[2]Department of Computer Science and Software Engineering, University of Canterbury, Private Bag 4800, Christchurch 8140, New Zealand

Abstract. This paper presents two well known design patterns that are appropriate for designing Interactive Simulation components for educational purposes. These are the Delegation Event Model, used for linking between components, and the MVC (Model-View-Controller) pattern, used for connecting the components to their visualizations and graphical user interfaces (GUIs). Combining both architectures, we have constructed Discrete Event Simulation (DES) components for modelling queuing networks in the Flash environment. The resulting components not only help teachers with little programming skill to construct simulation models, but also allow learners to conduct various experiments through interactive Graphical User Interfaces (GUIs) and obtain feedbacks of model behaviour through a range of engaging visualizations.

Keywords: DES, animation, component based modelling, design pattern, Flash simulator

1. Introduction

Ease of use and flexibility are essential criteria for Discrete Event Simulation (DES) tools. Unfortunately, both often conflict with each other. General-purpose DES simulators (e.g., *PSim-J* [1] and *SSJ* [2]) require significant programming effort for building models. Visual and interactive tools offer a user-friendly model construction environment. Unfortunately they often lack flexibility, since their architectures are hidden and difficult to extend with additional simulation logics.

Object oriented simulation libraries have long been used in providing a flexible simulation environment. However, 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 [e.g., see 3, 4].

Our primary focus is to design easy-to-use and extensible DES tools that foster *modelling for insight*; i.e., models that improve understanding through observation. Such models should incorporate interfaces to visualize model structures, activities to challenge learners' imagination and understanding, interesting scenarios to attract learners' activities, animation to depict model behaviours, and informative feedbacks to reflect learners' actions. All models should also be easily accessible. For this, we have used *Flash ActionScript* [5] since it offers robust support for component design besides its strengths as an animation tool and its support for cross-platform and integration with Learning Management Systems (LMSs).

This paper presents the concepts related to the design and development of DES components. It is organized into six sections. *Section 2* presents the basic principle of component-based simulation and surveys some existing component-based simulators. *Section 3* introduces the *Delegation Event Model* and examines its usefulness in forging links between DES components. *Section 4* presents the *MVC (Model-*

+ [1]ruzelan@uum.edu.my, {[2]wolfgang.kreutzer, [2]tim.bell}@canterbury.ac.nz

View-Controller) pattern and discusses how it can be utilized for loose coupling between components, their interfaces (GUIs) and their visualizations. The architecture of how both patterns can be implemented in the Flash environment is presented in *Section 5*. *Section 6* draws some conclusions.

2. Component Based Simulation

When describing his *DEVS* (Discrete Event System Specification) formalism, Zeigler [6] proposed that a simulation model should be built in a hierarchical and modular fashion; i.e., that a model is a collection of interconnected components. 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.

Each component is designed to guide message flows and to control such dynamic messages' movements. Messages are generated by the first "upstream" components and then transferred to other "downstream" components through output ports. Since downstream components are configured by upstream components, the only task of the downstream components is to react to messages they receive and update the messages' states. Based on this approach, many simulators have been built and reported; e.g., *XCELL+* [7], *SIMFACTORY* [8], *JSIM* [9], *Simkit* [10], *COST* [11], *Viskit* [12] and *BPSim++* [13].

Simkit and *COST* are not user-friendly since they only allow a designer to construct models through an *Application Programming Interface* (API). *XCELL+* and *SIMFACTORY* provide easy-to-use GUIs with which simulation models can be constructed by dragging components onto a canvas and connecting them. Since their internal architectures are hidden, 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. *JSIM* and *Viskit* are easy-to use and extensible tools with support for web-based simulation, but do not incorporate any visualization and animation facilities. Many modern simulation software (e.g., Arena [14]) meanwhile are excellent tools for building sophisticated simulation models and observing animation and visualization, however the capabilities to support user-directed experimentation during run time are limited.

3. The Delegation Event Model For Linking DES Components

The *Delegation Event Model* suggests a generic design for how to broadcast many different events (i.e., *event objects*) from an *event source* to all registered *event listener* objects. This style of event broadcasting is analogous to the flow of entities in DES systems, 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 the message is destroyed.

This design pattern plays two important roles in building DES simulators. Firstly, it avoids creating a class which defines an entity type's *lifecycle* method; i.e., a sequence of phases that all the entity instances must step through during their lifetime (e.g., using switch case statements). Writing such lifecycle descriptions become more complicated if entities need to be split (e.g., based on probabilities or conditions) at a certain phase of their lifecycles. Secondly, through sub-classing, other developers can extend our existing architecture to support new high level components (e.g., other simulation metaphors and styles).

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 component to component, using components' output ports. Our DES component development is based on the DES framework discussed by Khalid, Kreutzer and Bell [15], and the class and interface structures found in Moock [5] to build a suitable implementation in Flash ActionScript, which is illustrated in Fig. 1 (see [16] for a list of the DES components).

We use five basic classes and two interfaces to implement DES components (e.g., Source, Queue, Decide, Sink, etc.) based on this pattern; i.e., *ComponentSource*, *EventListenerList*, *Event Object*, *SimProcess*, *ComponentListener*, *EventListener* and *SimProcessListener*. *ComponentSource* (an *event source*) represents classes that schedule an instance of the *SimProcesss* class (a *SimProcess* object) and broadcast this object to all 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. *ComponentSources* can be equipped with a GUI to provide easy access points to its properties.

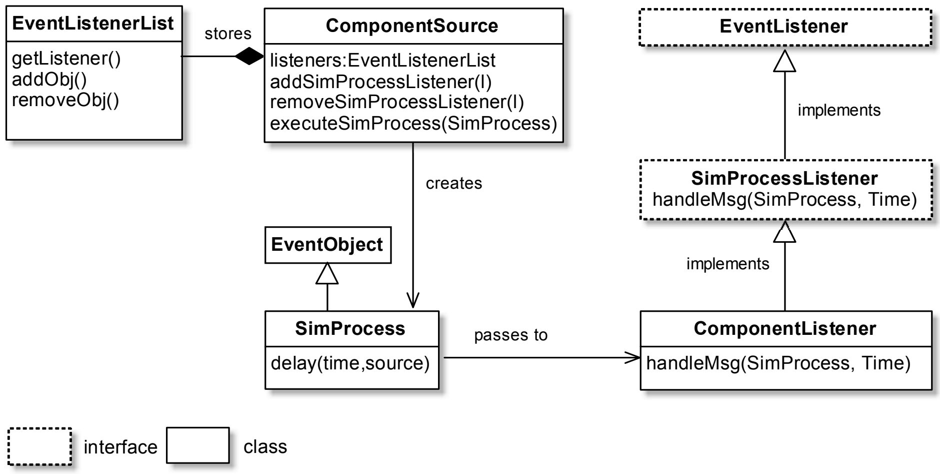
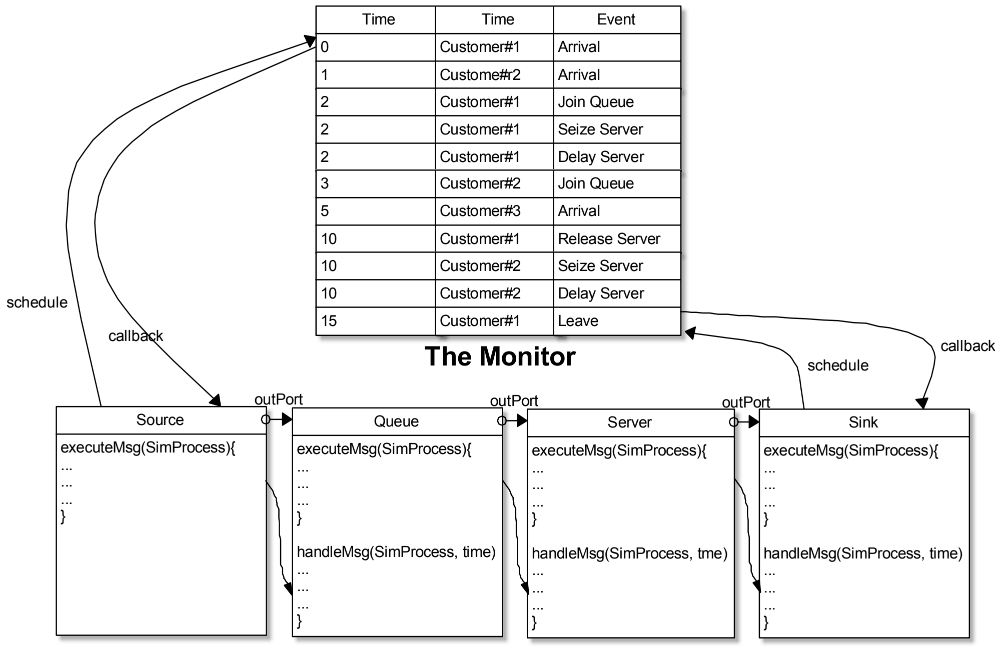


Fig. 1: The DES delegation event model structure

The *SimProcess* (an *event object*) class encodes entities that can be placed on an *Agenda* (a list that stores the next scheduled event for a particular *SimProcess* object) and will broadcast to *ComponentListener* objects when a 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*, 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 *ComponentSources*. When an event occurs the *ComponentSource* invokes a *handleMsg (SimProcess, Time)* method for each *ComponentListener* object.



Time	Time	Event
0	Customer#1	Arrival
1	Custome#r2	Arrival
2	Customer#1	Join Queue
2	Customer#1	Seize Server
2	Customer#1	Delay Server
3	Customer#2	Join Queue
5	Customer#3	Arrival
10	Customer#1	Release Server
10	Customer#2	Seize Server
10	Customer#2	Delay Server
15	Customer#1	Leave

The Monitor
schedule
callback
schedule
callback
outPort
outPort
outPort
Source
executeMsg(SimProcess){ ... ... }
Queue
executeMsg(SimProcess){ ... ... }
handleMsg(SimProcess, time) ... ... }
Server
executeMsg(SimProcess){ ... ... }
handleMsg(SimProcess, tme) ... ... }
Sink
executeMsg(SimProcess){ ... ... }
handleMsg(SimProcess, time) ... ... }

Fig. 2: The flow of a *SimProcess* object in DES components

Fig. 2 traces a simple flow of a *SimProcess* object in an M/M/1 queuing scenario. An instance of the *SimProcess* class 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. The *SimProcess* object is then 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) and broadcasts the *SimProcess* object to its all registered listeners by invoking *handleMsg (SimProcess, Time)*.

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 on the *Event List*. When the next scheduled time is reached, the *SimProcess* object calls the event source that scheduled it; the event source executes *executeMsg (SimProcess)* and broadcasts the *SimProcess* object to all of its downstream components. This is repeated until the *SimProcess* object departs from the system; i.e., when it arrives at a *Sink* - its lowest downstream component.

The main *tricky issue* in implementing this pattern in an animated simulator is to correctly trigger sorted events at appropriate times (i.e., to stop or delay events appropriately before triggering next events) based on a changeable runtime viewing ratio (i.e., a ratio of a given number of simulation time units into a corresponding number of seconds of animation time). To correctly delay the time between two consecutive events during animation, we multiply the interval of the delay time between two consecutive events with the inverse of the current viewing ratio. To smoothly transfer the entity so that it can reach its next component, we multiply the distance between the two components on a stage with the current viewing ratio and divide by its route time. Flash's *setInterval* and *clearInterval* functions are important to accomplish the task.

4. The MVC Pattern 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. There are three reasons why the *MVC* pattern is so useful for building interactive and attractive 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* 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 can be *reused* by different types of DES components.

Fig. 3 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, 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 class should have its own *ComponentUpdate* class with a unique name (e.g., *ServerUpdate*, *QueueUpdate*, etc.).

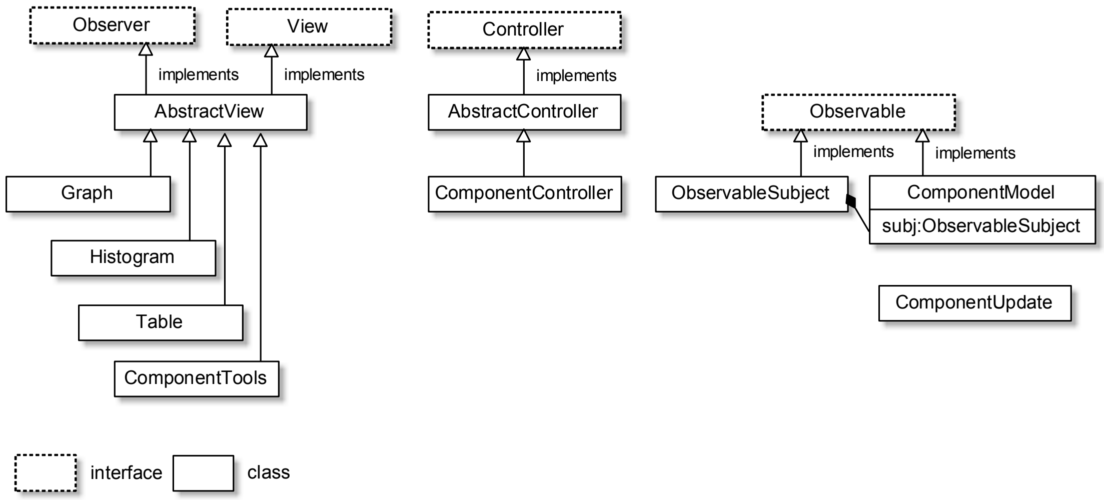


Fig. 3: The DES *MVC* structure

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 state changes. 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* must have its own controller (e.g., *ServerController*, *QueueController*). 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 changing states 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 this method to react to the information object sent by a *ComponentModel*.

5. Example

To demonstrate the ease of use of our DES components, we will develop a queuing network as in Fig. 4. 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, the entity needs to go to another queue before leaving the system. The second type chooses the shortest queue between the two available queues. Some percentage of the entities then exits the system while others need to go to the servers which process the first type of entity.

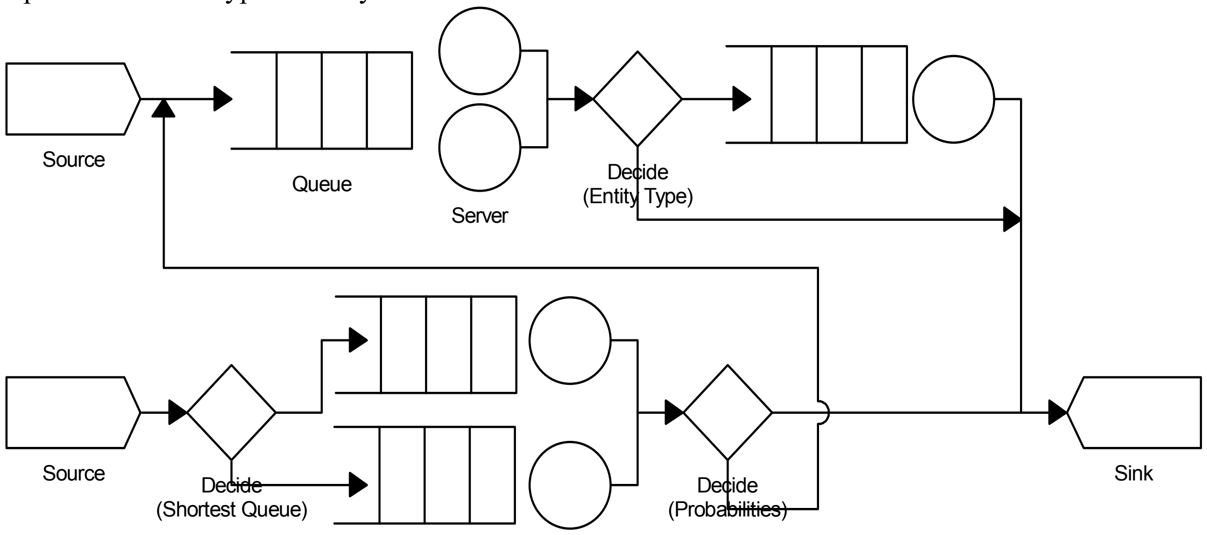
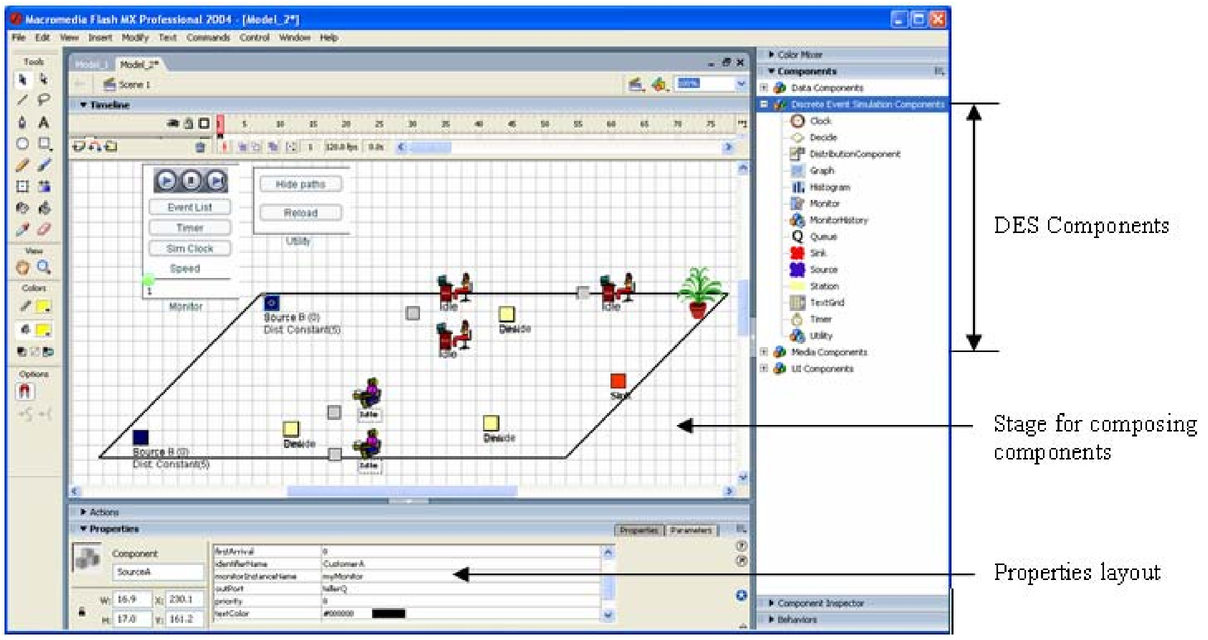
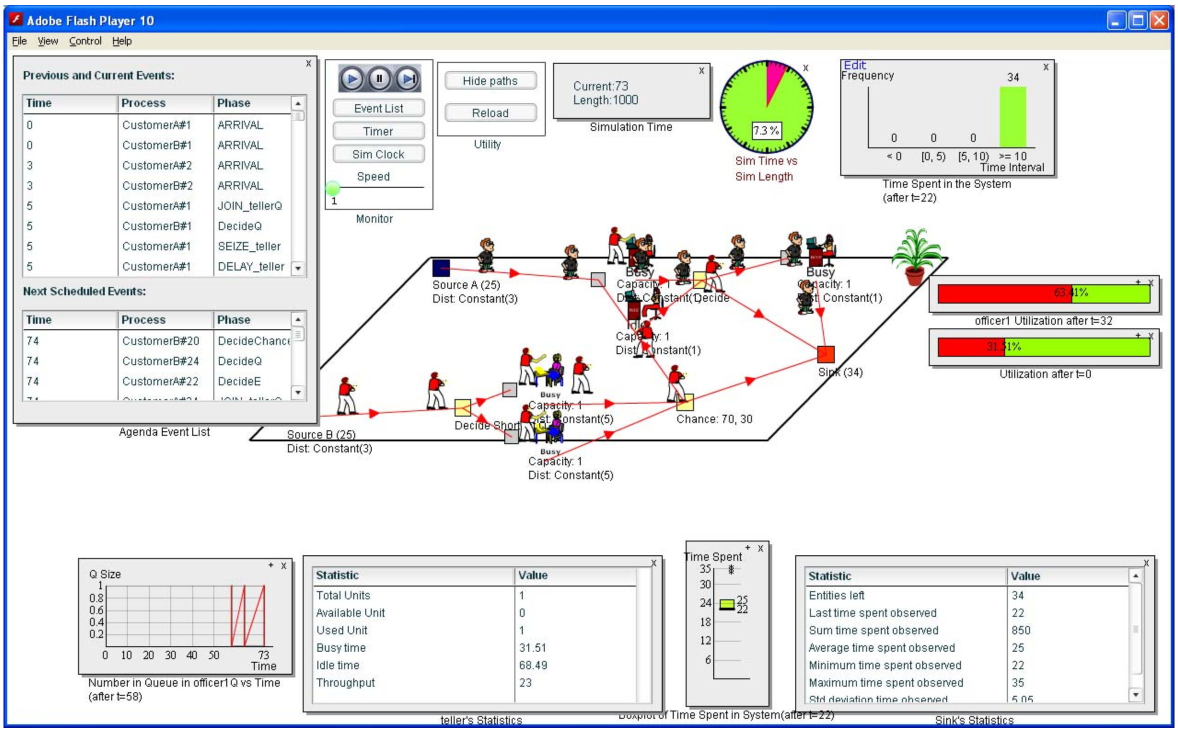


Fig. 4: A queuing network system

Based on this structure, 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. A *Monitor* instance is needed to coordinate the sequence of entities in a model so that entities can be transferred between components at appropriate times and in the right orders. All of these component instances need to be dragged and dropped onto the Flash's *Stage*, arranged accordingly and initialized their parameter values including their output ports; see Fig. 5(a). Fig. 5(b) shows a sample of the model that was constructed in this manner with its own customized visualizations. All data visualization can be customized and located at any location on the model stage.



(a) Development environment



Previous and Current Events:

Time	Process	Phase
0	CustomerA#1	ARRIVAL
0	CustomerB#1	ARRIVAL
3	CustomerA#2	ARRIVAL
3	CustomerB#2	ARRIVAL
5	CustomerA#1	JOIN_tellerQ
5	CustomerB#1	DecideQ
5	CustomerA#1	SEIZE_teller
5	CustomerA#1	DELAY_teller

Next Scheduled Events:

Time	Process	Phase
74	CustomerB#20	DecideChance
74	CustomerB#24	DecideQ
74	CustomerA#22	DecideE

Agenda Event List

Statistic	Value
Total Units	1
Available Unit	0
Used Unit	1
Busy time	31.51
Idle time	68.49
Throughput	23

teller's Statistics

Statistic	Value
Entities left	34
Last time spent observed	22
Sum time spent observed	850
Average time spent observed	25
Minimum time spent observed	22
Maximum time spent observed	35
Std deviation time observed	5.05

Sink's Statistics

(b) Runtime environment

Fig. 5: The model environment

6. Conclusions

In this paper, we have proposed two design patterns, the *Delegation Event Model* and *MVC*, for designing and building attractive and interactive DES components. Both patterns are well known, easy to understand and well documented and fit the needs of DES component design very well. Using these patterns in the context of building interactive and visual simulation modelling tool has resulted in components that can be combined to construct attractive simulation models and visualizations that can be easily animated and permit flexible interactions.

In order to obtain feedback from learners about the attractiveness, interactivity and usefulness of simulation models constructed with our components some experiments have been conducted. Data collected in this fashion is currently analysed to derive some hypotheses about the effectiveness of visual interactive simulation in a learning environment. We are also planning to distribute these components to teachers and seek feedback about their ease of use for model construction.

7. References

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

[2] P. L'Ecuyer, L. Meliani and J. Vaucher. SSJ: A framework for stochastic simulation in Java. *Proc. of the 2002 Winter Simulation Conference*. San Diego, 2002: 234-242.

[3] C. Alejandra, P. Mario and V. Antonio. *Component-Based Software Quality: Methods and Techniques*. Berlin: Springer, 2003.

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

[5] C. Moock. *Essential ActionScript 2.0*. Farnham: O'Reilley, 2004.

[6] B. P. Zeigler. *Multifaceted Modeling and Discrete Event Simulation*. London: Academic Press, 1984.

[7] R. Conway and W. Maxwell. Modeling asynchronous materials handling systems in XCELL+. *Proc. of the 19th Conference on Winter Simulation*. Atlanta, 1987: 202-206.

[8] K. Tumay. Factory simulation with animation: the no programming approach. *Proc. of the 1987 Winter Simulation Conference*. Atlanta, 1987: 258-260.

[9] J. A. Miller, Y. Ge, and J. Tao. Component-based simulation environments: JSIM as a case study using Java Beans. *Proc. of the 30th Conference on Winter Simulation Conference*. Washington, 1998: 373-382.

[10] A. Buss. Component based simulation modeling with SIMKIT. *Proc. of the 2002 Winter Simulation Conference*. San Diego, 2002: 243-249.

[11] G. Chen, and B.K. Szymanski. COST: A component-oriented discrete event simulator. *Proc. of the 2002 Winter Simulation Conference*. San Diego, 2002: 776-782.

[12] A. Buss, and C. Blais. Composability and component-based discrete event simulation. *Proc. of the 2007 Winter Simulation Conference*. Washington, 2007: 694-702.

[13] N. Melão, and M. Pidd. Using component technology to develop a simulation library for business process modelling. *European Journal of Operational Research*. 2007, 172(1): 163-178.

[14] W. D. Kelton, R. P. Sadowski and D. T. Sturrock. *Simulation with Arena.* New York: Mc-Graw Hill, 2004.

[15] R. Khalid, W. Kreutzer and T. Bell. Combining simulation and animation of queuing scenarios in a Flash-based discrete event simulator. *Proc. of UNISCON'2009*. Sydney, 2009: 240-251.

[16] R. Khalid, W. Kreutzer and T. Bell. Flash: Making simulations interactive. *Proc. of the SimTecT 2009*. Adelaide, 2009: 79-85.