Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

BUILDING COMPLEX MODELS WITH LEGOS
(Listener Event Graph Objects)

Arnold H. Buss

MOVES Institute
Naval Postgraduate School
Monterey, CA 93943-5000, U.S.A.

Paul J. Sánchez

Operations Research Department
Naval Postgraduate School
Monterey, CA 93943-5000, U.S.A.

ABSTRACT

Event Graphs are a simple and elegant language-independent way of representing a Discrete Event Simulation (DES) model. In this paper we propose an extension to basic Event Graphs that enables small models to be encapsulated in reusable modules called Listener Event Graph Objects (LEGOs). These modules are linked together using a design pattern from Object Oriented Programming called the "listener pattern" to produce new modules of even greater complexity. The modules generated in this way can themselves be linked and encapsulated, forming a hierarchical design which is highly scalable. These concepts have been implemented in Simkit, a freely available simulation package implemented in Java.

1 INTRODUCTION

In 1983 Lee Schruben first described Event Graphs, a simple, yet extremely powerful, construct to graphically portray DES models in a language-independent manner. An Event Graph is a graph (in the formal mathematical sense of being a set of vertices and edges) that captures the event logic of a given model. In an Event Graph the vertices represent the state transition function, while the edges capture the scheduling relationships between events (Schruben 1983).

Events Graphs are the most natural way to represent a DES model and, indeed, are the only such construct that directly captures the event-driven nature of such models. Although other constructs, such as Petri nets, can likewise model DES, Event Graphs provide the simplest and most elegant representation.

In this paper we describe how to extend Event Graphs to better accommodate large-scale DES models. The idea is to build small, manageable components that encapsulate Event Graph logic. These components correspond to the concept of an "object" in object-oriented programming: objects encapsulate state and behavior in a self-contained unit (Joines and Roberts 1999; Buss 2000). These component pieces can then be linked together to rapidly build larger, more complex models. The way in which this is accomplished is very important. In most programs two component pieces need to be tightly integrated to be able to interact successfully. If we used that approach, our components would not be self-contained. They would need to be modified for each type of use. To be able to use "off-the-shelf" components, one needs a way of establishing linkages and interactions to existing components without breaking their encapsulation. This is accomplished using an object-oriented design pattern (Gamma et. al. 1995) called the "listener pattern," or "listener" for short. Listener objects register their interest in other objects, and respond when they "hear" state changes. The choice of appropriate action, if any, lies solely with the listener. The object being listened to has no say in the actions which may be taken, and is not affected in any way, i.e., this is a non-invasive construction.

The end result of combining object encapsulation with the listener pattern is a language-independent modeling paradigm with all of the power of Event Graphs (which have been shown to be Turing complete, i.e., capable of modeling anything that can be programmed), but which also controls the complexity of large systems by allowing component design and reuse combined with hierarchical modeling. The obvious and natural name for this modeling paradigm is "Listener Event Graph Objects," with the acronym LEGOs. The name is also a metaphor for how complex models can be built by rapidly linking simpler component sub-models.

2 EVENT GRAPHS

In Event Graphs each event is represented as a vertex in the graph. By convention, each event vertex is given a descriptive label. All state transitions associated with the event are written either below the event and surrounded by curly braces ("{}"), or separately, indexed by the label.

Scheduling relationships between events are represented as directed edges. The edge originates at the event vertex performing the schedule operation and terminates at the

event vertex to be scheduled. Note that neither the vertices nor the edges need be distinct, i.e., a vertex is permitted to schedule itself, and there may be more than one scheduling relationship between a pair of events. Each scheduling relationship has an associated delay, which by convention is written above the line and at the originating end. If no delay is written, the value is assumed to be zero. Each scheduling relationship also has an associated condition, which by convention is drawn as a tilde shape near the midpoint of the edge. The condition is a boolean function of the system state. Scheduling occurs if and only if the condition is `TRUE`. If no condition is provided, the value is assumed to be `TRUE`. Finally, since the edge conditions are functions of the state, the order in which state transitions and event scheduling occurs is important. By convention, any and all state transitions associated with the event are completed prior to event scheduling activities.

Figure 1 illustrates these concepts. It can be translated to English as: When event A occurs, all state transitions associated with A are performed. Then, if condition c is `TRUE`, event B will be scheduled to occur t time units later.

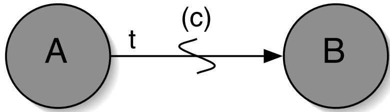

Figure 1: The Quintessential Event Graph

If the edge is a dotted line instead of a solid line, it represents cancellation. By convention, the next occurrence of the event being pointed to will be removed from the list of pending events. In theory, cancelling edges are not necessary. In practice, they can simplify the modeling process significantly.

Basic Event Graphs have been enhanced by the addition of formal parameters. Each event can have a formal parameter list associated with it. If this is the case, then all edges that schedule the event must have an associated list of arguments which match the parameter list in number, type, and order. By convention, the parameter list is written in parentheses after the event label, and the argument list is specified as a list of expressions contained in square brackets alongside the scheduling edge. The expressions are evaluated when the event is to be scheduled, and the resulting values are stored in the event list along with the event notice. Note that the parameters are *not* elements of the system state – the state can change between the time an event notice is scheduled and the time that event actually occurs. Parameters behave more like the arguments of a function or method – their scope is local to the event, and they store temporary copies of functions of the state, as computed at the time the event notice was placed on the event list.

Figure 2 demonstrates the addition of formal parameters. This graph can be translated to English as: When event A occurs, all state transitions associated with A are performed. Then, if condition c is `TRUE`, expression e will be evaluated to determine its value v and event B will be scheduled to occur t time units later with parameter p set to v.

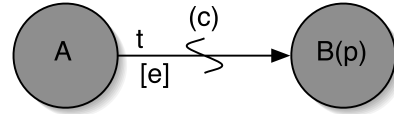

Figure 2: A Parameterized Event Graph

For more information about Event Graphs, including numerous examples of usage, we refer the reader to (Schruben 1983, 1992; Schruben and Yücesan 1993; Buss 2001a).

3 LEGOS

Listener Event Graph Objects (LEGOs) are fundamentally Event Graphs, but have two simple, yet important, additions. The first of these is encapsulation, and the second is the listener pattern.

3.1 Encapsulation

An Event Graph which represents self-contained functionality can be "wrapped" to create an Event Graph object, which can subsequently be treated as an atomic component in other models. We illustrate this visually by drawing a box or rectangle around the Event Graph. We offer two examples below.

3.1.1 The Arrival Process

The first example is a pure arrival process, illustrated in Figure 3. Arrive events schedule further arrivals after a delay of t_A time units. The state consists of a counter which tallies the number of arrivals which have occurred to date. This first example may not be terribly interesting, but note that once it has been formulated as an object the modeler can instantiate as many copies as desired, each of which maintains its own state and is parameterized by its own set of interarrival times $\{t_A\}$. When the Event Graph object is more complex than this simple first example, the benefits of code reuse can be significant.

3.1.2 The Multiple Server Queue

The second example is only slightly more complex. Figure 4 shows the Event Graph object for a multiple server queue. Readers familiar with Event Graphs will immediately recognize the model: An Arrive event increments the

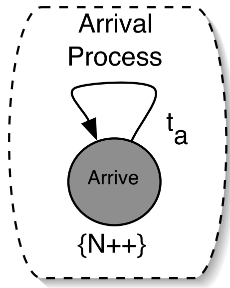

Figure 3: A Pure Arrival Process

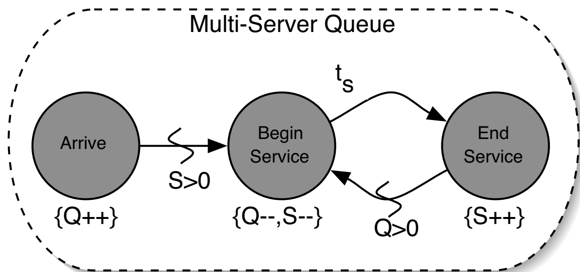

Figure 4: A Multiple Server Queue

number of customers who are waiting in line. If a server is available, a Begin Service event will be scheduled with no delay. When a Begin Service event occurs, the number of customers waiting in line and the number of available servers are both decremented, then an End Service event is scheduled to occur t_S time units later. When an End Service event occurs, the number of available servers is incremented. If there are customers awaiting service, a Begin Service event is scheduled with zero delay. As with the arrival process model, once a multiple server queueing Event Graph has been formulated as an object, the modeler can instantiate as many copies as desired, parameterized appropriately.

3.2 The Listener Pattern

In traditional Event Graphs, later events are actively scheduled by their predecessors. The relationship between a scheduling and a scheduled event is hard-wired into the model. The graph syntax is not designed to allow for the possibility that additional events might later be added to the model and triggered by the current event. To add additional scheduling relationships, the Event Graph must be modified by adding additional edges. Model elements are tightly integrated, which virtually destroys the potential for encapsulation and code reuse. We avoid this problem by using the listener pattern.

3.2.1 Linking Models with Listeners

The observant reader will have noted that the Arrive event in Figure 4 is not self-scheduling. The reason is that an arrival process that feeds a queue is not an integral part of the queue. For example, in a tandem queueing system the input process of a downstream queue is determined by the End Service events of the queue immediately preceding it. In order to accomplish this, we need a mechanism by which events occurring in one object trigger events in another object, without breaking the encapsulation. This is accomplished using the listener pattern. We illustrate this with a multiple server queueing model in Figure 5.

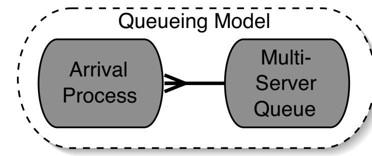

Figure 5: Linking Model Components with Listeners

The two Event Graph objects created in Figures 3 and 4 are linked by a new type of edge with an reversed triangle at one end, indicating a "listener" relationship. Think of it as a stethoscope: the listener edge indicates that the Multi-Server Queue object is registered to listen to the Arrival Process object.

The loosely-coupled nature of the listener connection is an important distinguishing feature of the LEGO framework. In the queueing model, the listener connection means that each occurrence of the Arrive event in the Arrival Process LEGO stimulates the occurrence of the Arrive event in the Multi-Server Queue. One simple way for this connection to be implemented involves "resonance," in which every event occurring in the listened-to LEGO triggers an event of the same name in the listening LEGO. If there is no corresponding event in the listener, then nothing happens. An important convention is that resonance does not propagate, i.e., if the Arrive event in the queue was triggered by resonance, it cannot in turn trigger other listeners by resonance. Only events which were actually scheduled within the component induce resonance in subsequent listener objects.

The listener pattern allows both the source and the listener to be unaware of each other in the sense that no specific knowledge is required. The Multi-Server Queue LEGO does not "know" that its Arrive event is being stimulated by an Arrival Process LEGO; all that is required to trigger the Arrive event is its occurrence in some LEGO it is listening to. Similarly, the Arrival Process LEGO has no awareness of the fact that its Arrive event is triggering an Arrive event in a Multi-Server Queue LEGO.

A crucial aspect of the listener pattern is that registering one LEGO component as a listener is *non-invasive* for

the LEGO component it is listening to. A LEGO component need not know anything about the context in which it is being used. This is analogous to the way in which chip designers use component designs to control the complexity of their task. Atomic components are assembled to sub-components, which are then combined into larger components. At the top level this yields circuit boards with a broad range of functionalities, which are nonetheless built from the same set of reusable off-the-shelf components. The circuit designer focuses on building linkages between components with known high-level functionality, rather than trying to rebuild each circuit from atomic components. A given component remains a sealed design, and has no information about the context in which it is being used. However, the outputs produced by that component become inputs to other components. The overall design resides in the linkage between the components.

3.2.2 A More Complex Example

In the prior example, linking the components together was simple and intuitive because an Arrive event in the Arrival Process corresponded to and triggered the execution of an Arrive event in the Multi-Server Queue. The situation is slightly different with a Tandem Queueing system – arrivals at downstream queues should be triggered by end of service events in the immediately prior queue, except for the first queue in the system. That one would receive its arrivals from an arrival process, as in the prior section.

The use of method name resonance introduces the need for a "bridge" (or "name mapping") object, i.e., an object which maps End Service events to Arrive events. This is illustrated in Figure 6. The bridge LEGO in Figure 6 is an extremely lightweight component. It maintains no state of its own and therefore has no state transitions. In other words, the bridge simply maps the event named End Service into the event named Arrive.

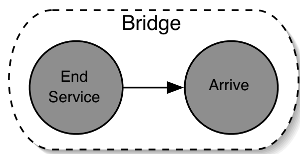

Figure 6: A Bridge Object

The tandem queue is then assembled from these component pieces. In Figure 7, three instances of the Multi-Server Queue and two instances of the bridge are instantiated. Each bridge acts as a listener to an upstream Multi-Server Queue instance and is listened to by a downstream instance of Multi-Server Queue. Note that the non-invasiveness of LEGO components is maintained in this example. Neither the Arrival Process nor the Multi-Server Queue has to be changed in the slightest to create the tandem queue model.

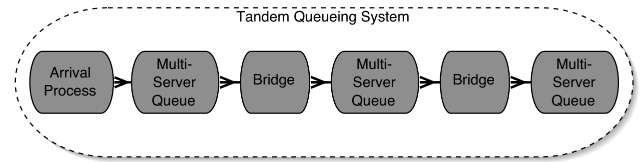

Figure 7: A Tandem Queue System

More complicated models can easily be envisioned using the listener pattern. For example, a single instance of an Arrival Process can be used to trigger many Multi-Server Queue instances. Similarly, arrivals that concatenate several kinds of processes can be easily modeled by instantiating multiple copies of the Arrival Process LEGO, each configured to capure the desired process, and the Multi-Server Queue can listen to all of them. An Arrive event in any of the Arrival Process instances will trigger an arrival to the queue.

4 HIERARCHICAL COMPONENT MODELING

The LEGO construct described above is sufficient to support hierarchical component modeling. The objective is to be able to construct LEGO components that themselves consist of other LEGO components so that the component can be used seamlessly without any knowledge of its internal construction.

To illustrate how this works, let us return to the tandem queueing model in Section 3.2.2. In that example, the components were the Arrival Process, the Multi-Server Queue instances, and the Bridge instances. Now we will encapsulate the entire model so that the internal LEGOs are shielded from the user and a simple Event Graph "interface" is presented to the outside world. The convention we use is that if a component name is the same as the name of the LEGO with "Interface" appended, that component is the one which listens to and/or is listened to by external LEGOs.

Following the convention just described, the hierarchical Tandem Queue Interface LEGO component contains two Event Graph pieces, as shown in Figure 8. The new model is not much more complicated than the Bridge LEGO illustrated in Figure 6. Like the bridge, the Tandem Queue Interface LEGO maintains no state of its own and performs no state transitions. It provides the externally visible interface to the componentized Tandem Queue model in Figure 7, excluding the Arrival Process. The first Multi-Server Queue LEGO is a listener to the Tandem Queue Interface, which in turn is a listener to the last Multi-Server Queue. An arrival to the Tandem Queue LEGO is triggered when the interface component hears a System Arrival event, which

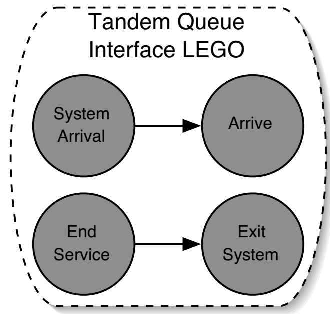

Figure 8: Tandem Queue Interface LEGO

schedules an Arrive event which in turn triggers the Arrive event of the first Multi-Server Queue instance. The End Service event of each Multi-Server Queue subsequently triggers the Arrive event of the subsequent queue via a bridge. When the End Service event occurs in the last Multi-Server Queue, it triggers the End Service event in the Tandem Queue Interface. This fires the Exit System event, which can be heard by external LEGOs. The design is illustrated in Figure 9.

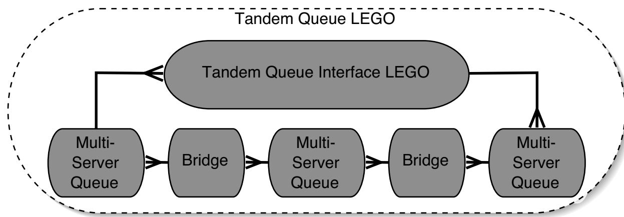

Figure 9: Improved Tandem Queue LEGO

Because of the interface component, the Tandem Queue LEGO looks to other components like the simple Event Graph shown in Figure 10. Instances of this LEGO can be combined with other components without having to know the details of the delay t on the edge in Figure 10, which is in fact generated by the behavior of the tandem queues.

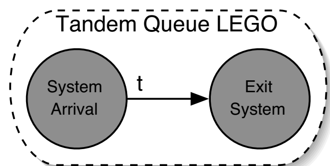

Figure 10: External View of Tandem Queue LEGO

The component design of tandem queue example presented above demonstrates the power of hierarchical component design that is possible using the LEGO framework. A LEGO component can be either atomic (like the Multi-Server Queue) or compound (like the Tandem Queue LEGO). Simulation modelers need not concern themselves with whether a given LEGO component is atomic or compound, but need only be concerned with the way the LEGO interacts with other components. All components have equal status and can be treated equally.

The LEGO framework, therefore, adds scalability to DES modeling in general, and to Event Graph modeling in particular. Atomic LEGO components tend to be relatively small, and thus easy to debug, verify, and maintain. The LEGO framework enables them to be assembled hierarchically into components with increasing features and complexity without sacrificing their maintainability. Attempts to build large monolithic models result in complex models that are difficult to modify or maintain. The LEGO framework enables much more large-scale models to be developed.

5 APPLICATION TO DATA

The listener pattern described in Section 3.2 proves to be extremely effective when applied to data and output analysis of simulation models. The loose coupling of the LEGO components means that information about state changes can be transmitted and received so that the model dynamics can be separated from the data collection. This provides an important feature to LEGO models, namely the fact that models can be created without having to worry about the particular statistics that will be desired, or even what other ways run data might be displayed. As long as every state transition fires an event that can be listened to, then no LEGO component need contain any logic whatsoever for collecting or logging data. New data-gathering schemes may be easily implemented. Most important, new data gathering can be implemented non-invasively.

For example, suppose that the time-averaged number in queue for a run is to be estimated for the queueing model above. To accomplish this, a listener object that updates statistical counters is instantiated and registered as a listener to the Multi-Server Queue instance. When this statistical listener receives the event signaling the change of value for the number in queue state (Q), it simply updates its counters. Whenever desired, it can be queried about the current value of the desired average.

Now suppose that we also desire to plot the state trajectory for the number in queue. This can be accomplished by instantiating a graphical listener and also registering it to listen to the Multi-Server Queue instance. Whenever it hears an event signaling the change of state of Q, it adds the new data point and redraws the plot. In a similar manner, a logging listener can write the state information and time of change to a log file.

Since there is no restriction on how many listeners a LEGO component can have, there is no restriction on the types of data that may be collected simultaneously. The

analyst has complete flexibility and control over how the model run is observed. The non-invasive nature of this manner of listener data gathering ensures the integrity of the data. If the model code must be edited or changed to gather new data, then there is always a risk that error may be inadvertently introduced into a previously correct model.

6 IMPLEMENTATION

The LEGO framework has been implemented in Simkit (Buss 2001b, 2002), a package written in Java for building DES models. An Object-Oriented language such as Java is particularly well-suited to implement a loosely-coupled component framework such as LEGO.

However, the LEGO framework is not tied to either Simkit or Java. Although the designs here were inspired by Object-Oriented modelling, and are very straightforward to implement in languages such as Java, LEGOs are a general purpose modeling tool and may be implemented in most any modern computer language. For example, the listener pattern can be implemented in procedural languages using a registration/call-back scheme. However, unless the programmer is using a language designed to incorporate separate name spaces, such as Modula-3 or Ada, care must be taken with the implementation to avoid name collisions for the events.

7 CONCLUSIONS

Using the techniques described in this paper, simulation practitioners can rapidly and accurately build models of complex systems. LEGO models control complexity by incorporating hierarchical design and encouraging reuse of previously designed and well-tested components. The components are linked non-invasively so that they benefit from encapsulation. The listener pattern used to create the linkages can be used for both model construction and data collection. These techniques are not just theoretical abstractions, they have been successfully implemented in the Simkit package developed at the Naval Postgraduate School.

ACKNOWLEDGMENTS

We would like to acknowledge Lee Schruben for his original contribution in creating Event Graphs, and for sharing his insights and ideas about modeling over a course of many years. We would also like to thank Susan M. Sanchez and Richard Mastowski for their invaluable feedback on early drafts of the paper. The authors were supported in this work by a grant from the Air Force Office of Scientific Research. This support is gratefully acknowledged.

REFERENCES

Buss, A. 2000. Component Based Simulation Modeling. *Proceedings of the 2000 Winter Simulation Conference*, ed., J. A. Joinses, R. R. Barton, K. Kang, and P. A. Fishwick. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Buss, A. 2001a. Basic Event Graph Modeling. *Simulation News Europe*. 31:1-6.

Buss, A. 2001b. Event Graph Modeling with Simkit. *Simulation News Europe*. 32/33:15-25.

Buss, A. 2002. Component-Based Simulation Modeling Using Simkit. *Proceedings of the 2002 Winter Simulation Conference*, ed., E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Joines, J. and Roberts, S. 1999. Simulation In An Object-Oriented World. In *Proceedings of the 1999 Winter Simulation Conference*, ed., P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Schruben, L. 1983. Simulation Modeling with Event Graphs. *Communications of the ACM*. 26: 957-963.

Schruben, L. 1992. *Sigma: A Graphical Simulation Modeling Program*. The Scientific Press. San Francisco, CA.

Schruben, L. and E. Yücesan. 1993. Modeling Paradigms for Discrete Event Simulation. *Operations Research Letters*. 13: 265-275.

AUTHOR BIOGRAPHIES

ARNOLD H. BUSS is a Research Assistant Professor in the MOVES Institute at the Naval Postgraduate School. His interests include component-based simulation modeling, with emphasis on military applications. His e-mail and web addresses are `<abuss@nps.navy.mil>` and `<http://diana.or.nps.navy.mil/~abuss>`.

PAUL J. SÁNCHEZ is in the Operations Research Department at the Naval Postgraduate School. His research interests include component-based simulation modeling, object-oriented modeling, and simulation output analysis. He is an avid reader and collector of science fiction, and enjoys riding recumbent bikes around the Monterey Bay area. You can reach him by e-mail at `<PaulSanchez@nps.navy.mil>`, and his web address is `<http://diana.or.nps.navy.mil/~pjs>`.