VISUALIZATION OF EVENT EXECUTION IN A DISCRETE EVENT SYSTEM

Samantha C. Collins
Nathan D. Gonda
Lee C. Dumaliang
James F. Leathrum, Jr.
Roland R. Mielke

Department of Modeling, Simulation and Visualization Engineering
Old Dominion University
5115 Hampton Blvd, Norfolk, VA, USA
{scoll003@odu.edu, ngond002@odu.edu, lduma002@odu.edu,
jleathru@odu.edu, rmielke@odu.edu}

ABSTRACT

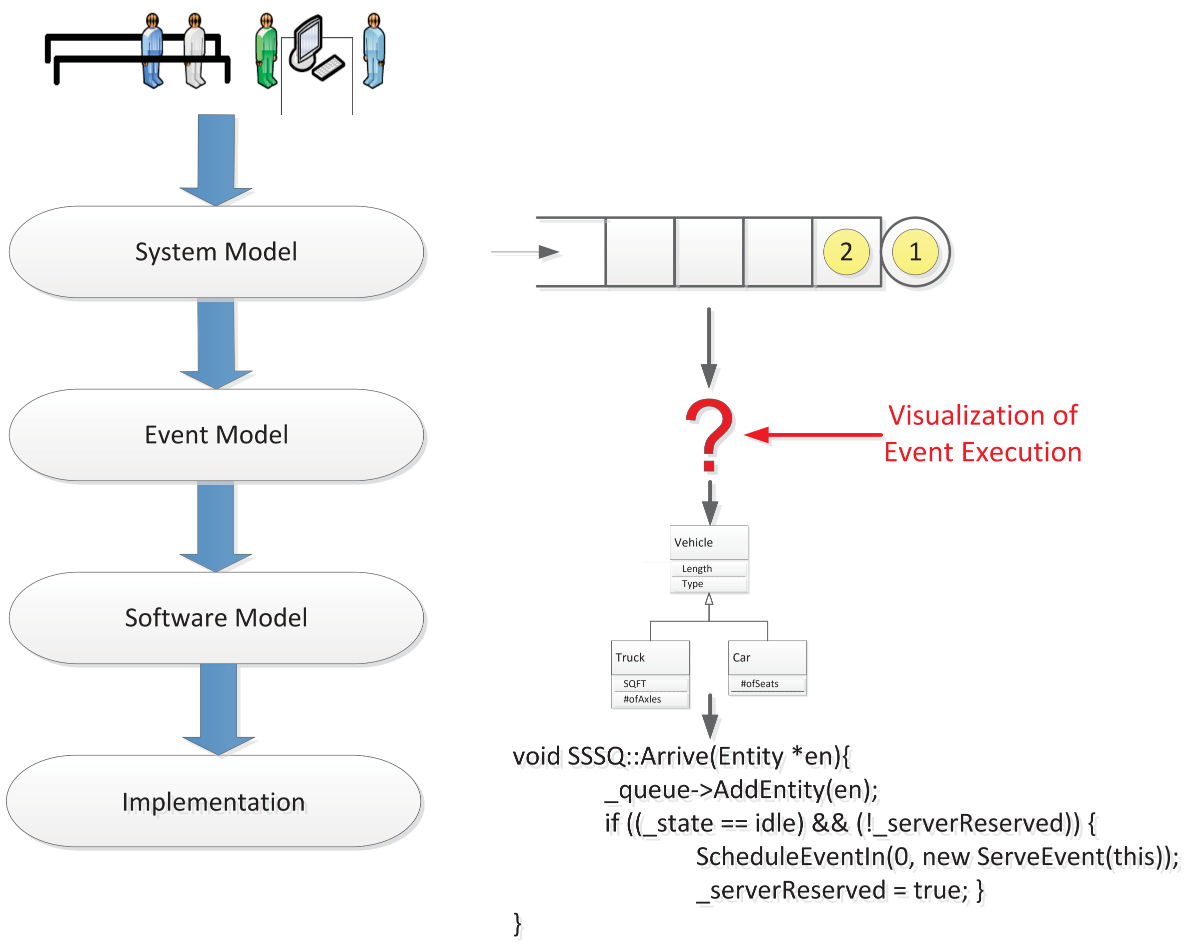
In discrete event simulation development, a large conceptual leap often is made when going from the system model to the simulation implementation. The event model, describing how events are scheduled and executed, is implied rather than explicitly stated. This paper addresses the gap by introducing a new visualization tool based on event graphs. Given an event graph representation of the event model, the tool allows the developer to observe the execution and scheduling of events during simulation execution. The use of the visualization tool encourages students to explicitly consider the event model, thus providing additional insight to the relationship between the model and the implementation. To better support more general discrete event simulation development efforts, including software development and verification and validation activities, future research will consider interfacing the visualization tool with simulation software tools and developing the capability to reverse engineer the event model from an existing simulation.

Keywords: discrete event simulation, visualization, simulation software, event graph.

1 INTRODUCTION

It is well understood that modelers and simulation developers need to understand the underlying mechanisms that simulation software uses to manage the simulation, such as how simultaneous events are handled (Schriber, Brunner, and Smith, 2016). Currently, the standard method for supporting simulation development in observing the low level effect of these mechanisms is through the use of event traces (Schriber, Brunner, and Smith, 2016**).** This paper presents a new software tool that provides a visualization of the management of events through the use of event graphs (Schruben, 1983).

The modeling and simulation development process typically progresses from developing a system model based on the system under study, to developing an event model that implements the system model, and finally to implementing the simulation, as illustrated in Figure 1. The simulation implementation commences with the development of a software model that is then implemented using a simulation tool, a simulation or general purpose programming language, or even an alternative environment such as a spreadsheet. Visualizations driven by the simulation usually tend to visualize either the system under study or the system model. It is suggested here that a visualization of the event model is both advantageous for the simulation developer and for use in modeling and simulation education. The authors believe that the event graph is an appropriate model of how the simulation implements and manages events, thus filling this need.



```
void SSSQ::Arrive(Entity *en){
        _queue->AddEntity(en);
        if ((_state == idle) && (!_serverReserved)) {
                ScheduleEventIn(0, new ServeEvent(this));
                _serverReserved = true; }
}
```

Figure 1: High level view of discrete event simulation development.

The approach taken to visualize the behavior of events is to develop a separate visualization tool with the necessary hooks to drive the visualization from a variety of simulation environments. The visualization includes an event graph that is highlighted to indicate the current state of execution, and a linear event trajectory highlighting the history of event execution and the known future events. The visualization tool provides the capability to control the advancement of the simulation execution, allowing the developer to step through the simulation.

The visualization tool was developed for educational purposes to support a course in discrete-event simulation that utilizes simulation tools and spreadsheets, and a course on software design for discrete-event simulations where simulations are developed in a general purpose programming language. As a result, the initial implementation of the visualization tool is scaled for the class of problems usually studied in an undergraduate academic environment. However, if the use of the visualization tool proves successful, there are many ways it might be enhanced for the development and verification of larger scale simulations.

The paper is organized in seven sections. In Section 2, the use of the event graph and the associated visualization of event execution are described. Then, in Section 3, related work is presented that supports the use of the visualization concept. The software architecture developed to facilitate integrating the visualization tool with various simulation development environments is presented in Section 4. An example to illustrate how the visualization tool displays the actions associated with the updating of state variables and the scheduling of future events is given in Section 5. Initial results are presented in Section 6 based on fielding the tool in a simulation software design course. Finally, Section 7 identifies future enhancements and extensions to the visualization tool.

2 APPROACH TO EVENT SCHEDULING AND EXECUTION VISUALIZATION

The focus of this paper is on providing a visual representation of the scheduling and execution of events in a discrete event simulation. The approach taken is to represent the event model using the combination of an event graph (Schruben, 1983) and an event set. The event set consists of three subsets: a set of previously executed events, a set containing the currently executing event, and a set of pending events. By updating this view during simulation software execution, each time a new event is scheduled and each time a new event is executed, a clear view of the management of events is presented to the simulation developer with the associated activity highlighted. To facilitate the updating of the visualization event set, simulation events are considered to consist of a set of actions. A single action accounts for the updating of the state variable values. An additional action is defined for each scheduling of a future event.

Event graphs clearly display the scheduling relationships between events in the system. An example event graph for a simple G/G/1 queue is shown in Figure 2. Nodes in the graph represent events in the system and edges represent the scheduling of a future event with the edge weight denoting how far in the future the event is scheduled. Edges also can have conditions associated with them indicating the conditions for which a future event is scheduled..

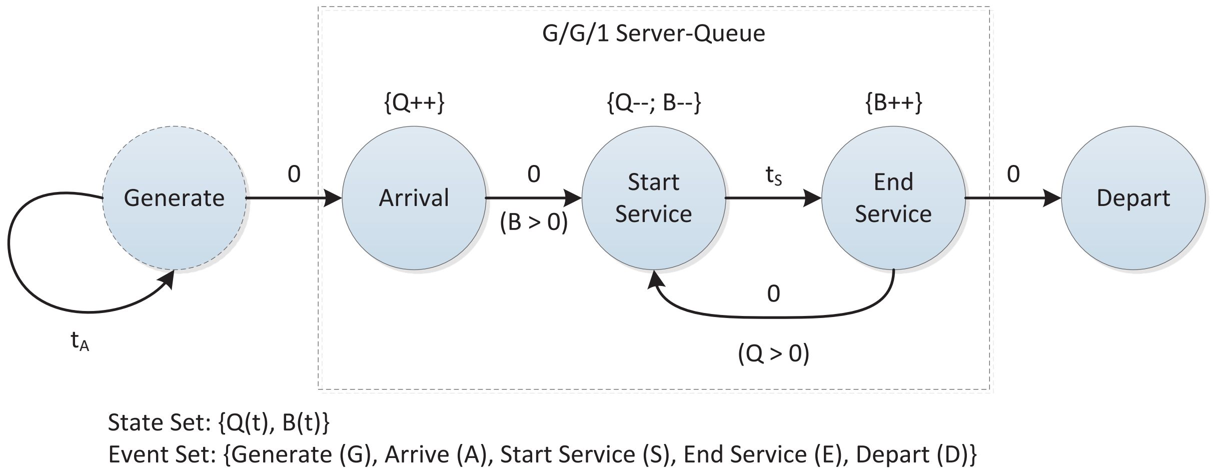


Figure 2: Example event graph.

Nodes also can be assigned notation indicating the change in state variable values caused by that event. For this example, the state variables are the number of available server resources (B(t)) and the number of entities in queue (Q(t)), where ‘t’ is simulation time measured from the beginning of the simulation. In Figure 2, the events have the following behavior as indicated in the event graph:

- Generate (G): An entity is created and presented to the G/G/1 queuing system as an input. Schedule an Arrive (A) event with a delay of zero time units; Schedule a Generate (G) event with a delay of t_A time units.
- Arrive (A): An entity arrives at the G/G/1 queue. Increment Q by one (Q++); If a server resource is available (B > 0), schedule a Start Service (S) event with a delay of zero time units.

- Start Service (S): An entity moves from the queue, seizes a server resource, and begins service. Decrement the queue by one (Q--); Decrement the available server resources by one (B--); Schedule an End Service (E) event with a delay of t_S time units.
- End Service (E): The server completes service, the entity releases the server resource and exits the G/G/1 queuing system. Schedule a Depart (D) event with a delay of zero time units; Increment the available server resources by one (B++); if $Q > 0$, schedule a Start Service (S) event with a delay of zero time units.
- Depart (D): An entity exits the G/G/1 queuing system.

Figure 3 demonstrates that the same system can be implemented using an alternate definition of the set of events, requiring a different visualization of its execution. In this example, the Start Service event is removed. Therefore, the Arrival and End Service events must be redefined to directly schedule the next End Service event.

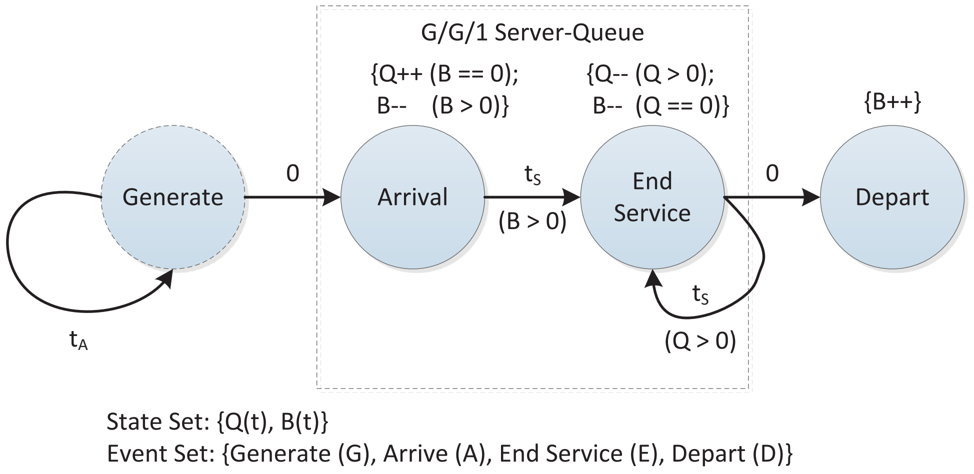


Figure 3: Example alternate event graph.

The event graph provides a convenient method of illustrating the currently executing event by simply highlighting that event on the graph and indicating the simulation time value associated with that execution. The sequence of scheduling other events for future execution is illustrated by highlighting the edges one at a time. What the event graph does not provide is insight into the sequence of events to be executed in the future, or the trajectory. In addition, it is considered insightful to see the trajectory that produced the current simulation state by showing the sequence of previously executed events. All of this is achieved with the event set.

The event set is defined as

$$event\ set = \{\{set\ of\ previously\ executed\ events\}, \{current\ event\}, \{set\ of\ pending\ events\}\}$$

On initialization of the simulation, the event set is defined as $\{\{\},\{\},\{\text{initial set of pending events}\}\}$, where $\{\}$ denotes the empty set. The event set is visualized as an ordered sequence of events, each event defined by its name and its execution time as shown in Figure 4. The three regions of the event set are color coded to differentiate the three subsets. As a new event is executed, the previously executed event is moved to the set of previously executed events, the next event in the set of pending events is moved to the current event, and the appropriate node is highlighted in the event graph. As a new event is scheduled and its edge highlighted in the event graph, the new event is inserted into the set of pending events in sorted order. It should be noted that the visualization tool does not have knowledge of tie breaking strategies for simultaneous events. Events in the set of previously executed events with identical times are ordered based on their execution order. In the set of pending events, events are ordered based on the order with which they are added to the set. However, this may not be the order they are executed. Therefore, when events have identical times that equal the current simulation time, the first in line may not be the first moved to the current event. Rather, the visualization tool matches the executed event to the appropriate pending event, potentially reordering the events as they transition from pending to previously executed event subsets.

Figure 4 shows the visual display produced by the visualization tool. This display is a combination of the event graph and the visualization event set. In this example display, the currently executing event is an End Service event executing at time 5, color coded green in the display. There is a past history of a Generate, Arrival, and Start Service events, all of which executed at time 0, followed by another Generate and Arrival executed at time 5. There are currently three pending events, Departure and Start Service events scheduled to execute at time 5 and a Generate event scheduled for time 10. The example is highlighting the action of the End Service event scheduling a Start Service event to execute at time 5. To do so, the associated edge between the events in the event graph is highlighted, and the scheduled event in the set of pending events is highlighted. To accommodate event sets having many elements, the visualization display is windowed to show only a portion of the entire event set. In general, the current event is centered on the display, producing a view of those events recently executed and those scheduled to execute in the near future, though the tool supports sliding the view to either side. Not shown is that the visual includes a console window. When updating the visual, the developer can supply a string that is presented in the window, allowing them to provide state variable information, parameters to events, or general debugging information.

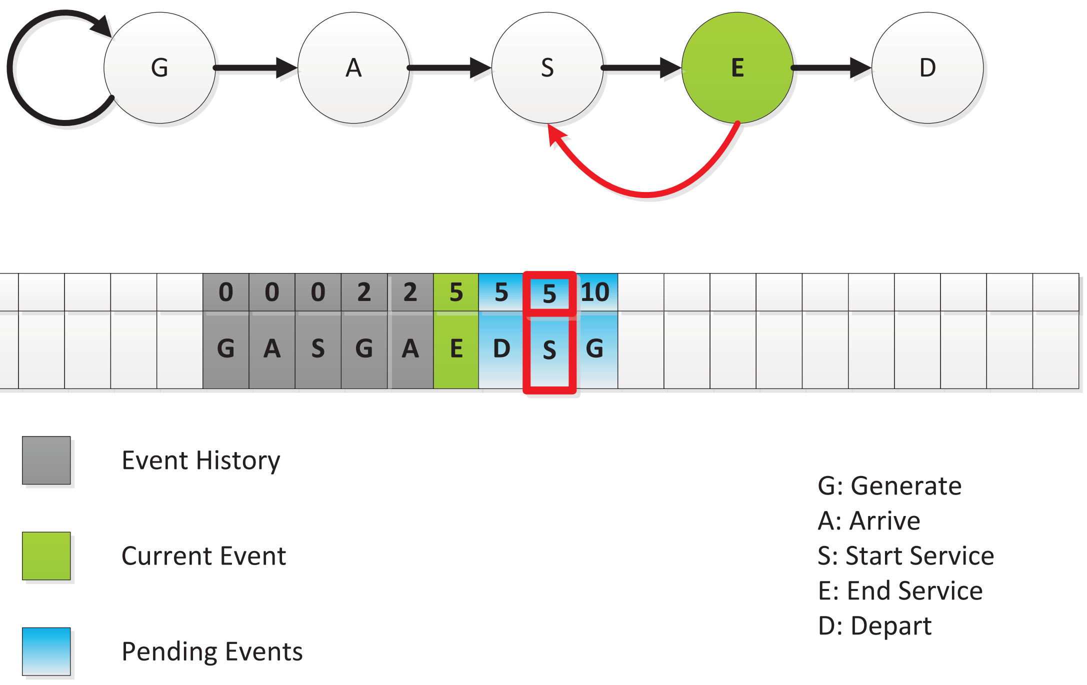


Figure 4: Visualization layout.

3 RELATED WORKS

The concept of having access to the underlying event behavior is not new. Schriber, Brunner, and Smith (2016) discuss the concept of interactive model verification. This involves introducing breakpoints into the software or running an event trace so that event information is output to the developer to assist in verifying that the simulation behaves as the model specifies. These capabilities are common in discrete-event simulation tools. When combined with a visualization of the system model, the difficulty is that there often is a layer of abstraction between the model and the underlying event model that generates the timing of event activity. Therefore, the developer is directly comparing the simulation's event actions to the high level model. This paper proposes introducing a low level event model visualization using event graphs.

Schruben (1983) introduced the concept of event graphs as a high-level model for discrete-event systems. Event graphs provide insight into the relationships between events, but have garnered little acceptance as a general modeling approach as the graphs are disconnected from the general thinking that modelers usually follow. However, the utility of event graphs can be found in their capability to characterize the underlying behavior of discrete event simulations. For example, event graphs have been found useful in proving the equivalence of event models. (Yucesan and Schruben, 1992; Bergen-Hill and Page, 2010) This allows the behavior of high level models that have been mapped to event graphs to be compared. This paper puts forth that this lower level modeling of discrete event simulations is where the real benefit of event graphs arises. The event graph is so closely related to the scheduling and execution of events that it is best used to visualize this low level activity.

SIGMA (Simulation Graphical Modeling and Analysis) (SigmaWiki, 2016; Schruben, 1992) is a discrete event simulation tool based on event graphs. A user defines an event graph by defining each event, the changes that event makes to the state variable values, and the future events scheduled due to the event occurrence. The event graph then can be simulated, allowing the user to single step through the simulation to observe the event behavior. However, the SIGMA tool does not allow the visualization of events generated in other software systems.

IBM Rational Rose Modeler is an example of a tool that provides insight into the relationship between the software model and the software implementation. It is an object-oriented UML software design tool used for visual modeling and component construction of large-scale software applications. A user creates UML diagrams which are documented and used to generate code. The software was designed to enhance software design and development by emphasizing the importance of conceptual models and modular software architectures. It also provides a visual representation of the execution of the model in conjunction with the software execution. The purpose of this paper is the development of a similar capability in discrete event simulation.

Similar research on visualizing a scheduled sequence of activities, past, present, and future, has been conducted on the capability to visualize CPU schedulers. Suranauwarat (2007) presents a simulator with graphical animation intended to increase understanding on CPU scheduling algorithms. The simulator's animation contains a graph showing status and a separate view of corresponding color-coded timelines. These timelines consist of labeled blocks with time indicated above. This simple yet effective representation demonstrates the utility of the authors' approach to other problem domains.

4 VISUALIZATION TOOL ARCHITECTURE

The event graph visualization tool presented here is designed as an add on visualization of a discrete event simulation developed in a separate environment, either a simulation developed in a software language, a simulation tool, or even a spreadsheet simulation. To that end, the visualization tool is designed as a separate piece of software in which an interface is developed to integrate the actual simulation. The visualization tool updates its view of the simulation execution at event actions. The simulation must communicate to the visualization tool when an event action occurs and indicate the nature of the action. The visualization tool must update the visualization display and then communicate to the simulation that it can advance to the next event action. This message interchange allows the visualization tool to control the advancement of the simulation. This can be done by single stepping through the actions, running through the simulation at a specified pace, or running through the simulation as fast as possible.

The visualization tool's architecture is related to the model-view-controller (MVC) design pattern approach to graphical user interface (GUI) design. A good survey of the current state of the MVC pattern is provided in Karagkasidis (2008). This software design pattern separates the model data from its presentation and interaction with the user. The MVC pattern is often used for GUIs and similar applications. A modified version of this architecture was a natural choice for this tool.

Figure 5 displays the software architecture integrating the visualization tool with a simulation developed in an arbitrary environment and highlights the interaction between the two systems. Decoupling the two systems and providing a communication mechanism between them facilitates integrating the visualization tool with different simulation environments. Currently, interfaces have been developed for C++ simulation code and Excel spreadsheet simulations because both are used in our educational environment.

The architecture assumes that the simulation application is capable of identifying the points at which an event execution is initiated and when an event schedules new events. It does not assume access to the state variable space. On instantiation of the visualization tool, the definition of the event graph is provided to the tool. It is the responsibility of the simulation developer to ensure this definition faithfully represents the application's event model. When either an event is executed (associated with the update of the state variables) or scheduled, the application requests an update to the simulation tool display through an API. This is passed on to the visualization tool through a message at which point the display is updated. While this is happening, the API retains control of the simulation, forcing it to wait until the visualization tool grants permission to continue by sending a continue message. The visualization sends a continue message indicating the visualization mode, single stepped, paced, or as fast as possible. When the API receives the continue message, it passes execution control back to the simulation application which continues until the next event action.

Two features of the visualization are of note here. First, in addition to updating the display view, the update messages may include a string that the developer passes on for display. This allows the developer to display other information, for instance to indicate state variable changes. The second feature is that the visualization is capable of performing basic error detection. For example, should the simulation attempt to schedule an event that is not properly indicated by an edge in the event graph, the visualization will flag an error.

5 EXAMPLE

In this section, a brief example is presented to illustrate the graphic displays produced by the visualization tool. The system model for this example is the G/G/1 queuing system that is described in Section 2. The event set for this model consists of five events: Generate (G), Arrive (A), Start Service (S), End Service (E), and Depart (D); the state set consists of two state variables: Q(t) = number of entities in the queue, and B(t) = available server capacity. It is assumed that the simulation of the model is represented by the event graph shown in Figure 2.

It should again be noted that an event may consist of several actions. These actions include (1) updating the value assigned to state variables, and (2) scheduling future events. The visualization tool views the updating of all state variables to be a single action while each scheduling of a future event is viewed as a separate action. The simulation must send a request update message to the visualization tool after each action for each event. The majority of discrete event simulation tools are able to support this requirement through one of several methods.

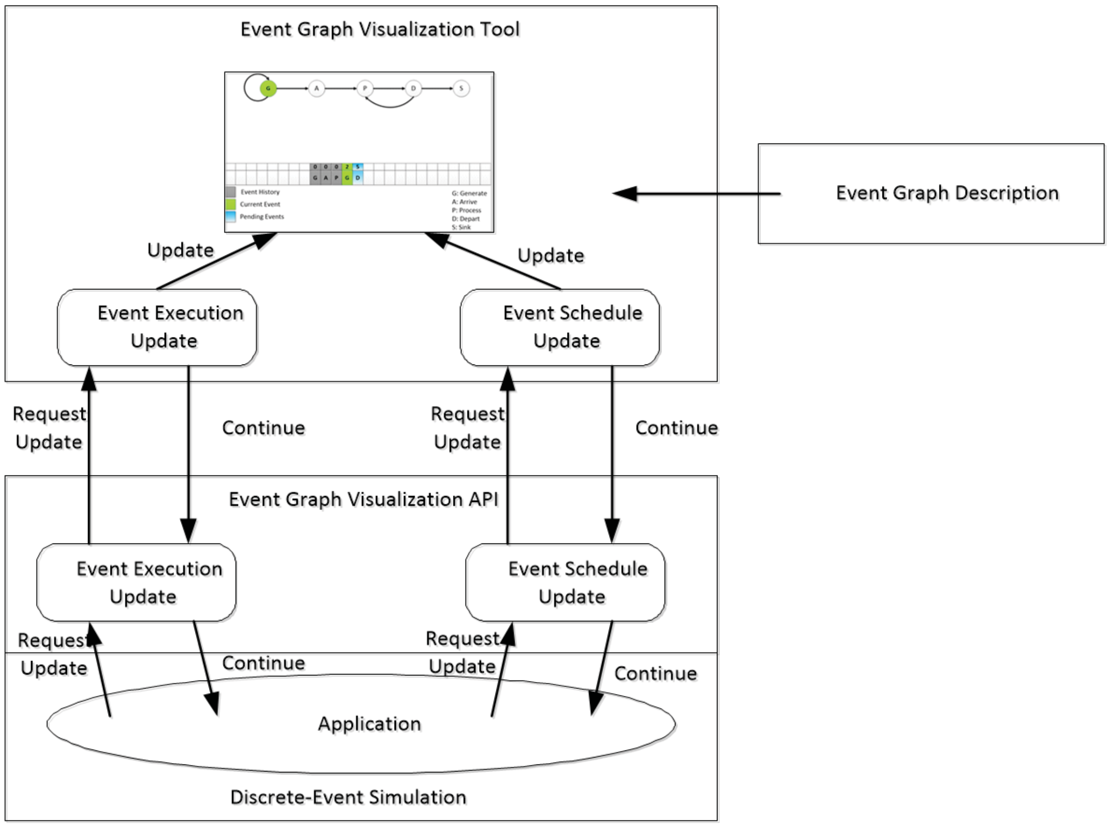
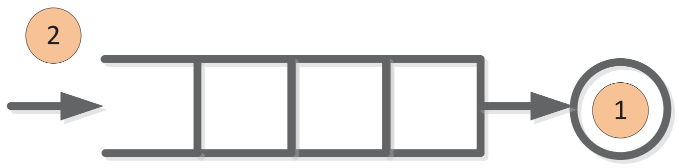


Figure 5: Software architecture.

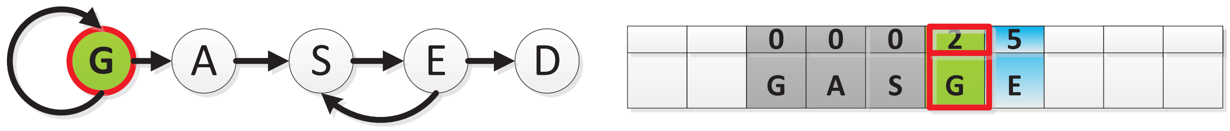
The example begins at simulation time t = 2 with the second occurrence of a generate event. The system simulation actually started empty and idle at t = 0. The first entity was generated, arrived at the queue, and then started service, all at t = 0; this entity is scheduled to end service at t = 5. As the example begins, the first action, state variable update, of the second Generate event is executing. The G/G/1 queuing model has state variable values Q(2) = 0, B(2) = 0 as shown in Figure 6a. The visualization tool display, consisting of the corresponding event graph and the associated time-stamped event set, is shown in Figure 6b. The Generate event is colored green to indicate that it is the current event. The events Generate, Arrive, and Start Service are shaded to indicate that they are previously executed events, and the event End Service is colored blue to indicate that it is a future scheduled event. This display will persist until a request update is passed from the simulation to the visualization tool.

As shown in Figure 6c, following the request update message, the visual display updates to show the result of the second action of the Generate event. The Generate event remains colored green to indicate that it is still the current event. The event graph edge directed from the Generate event to the Arrive event is colored red to indicate that a future Arrive event is being scheduled and the Arrive event is entered into the pending event list with a time increment of 0 time units. A continue message is then issued to the simulation in order to resume the execution process.

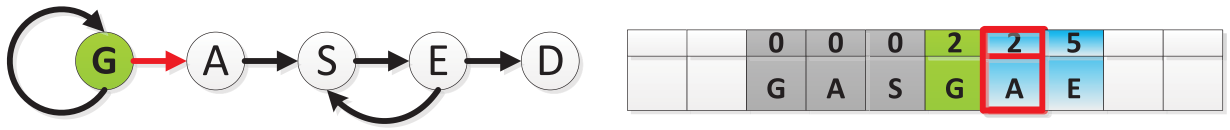
In Figure 6d, following the next request update message, the display updates to show the result of the third action of the Generate event. The event graph edge directed from the Generate event to the Generate event is colored red to indicate that a future Generate event is being scheduled and the new instance of the Generate event is entered into the pending event list with a time increment of 8 time units. This display is shown in Figure 6d and persists until the next request update message is issued. This action completes the execution of the Generate event.



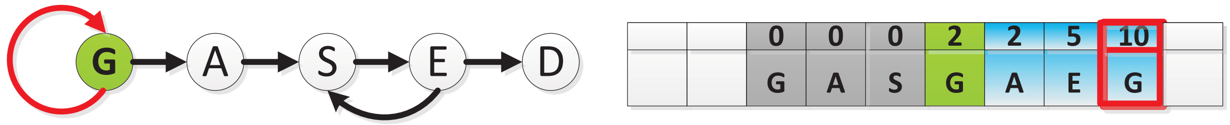
(a) Initial State of the Queue



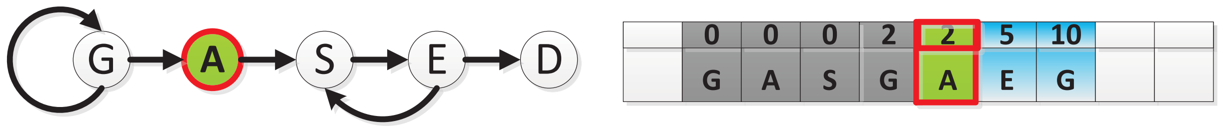
(b) Execution of Event G at time 2



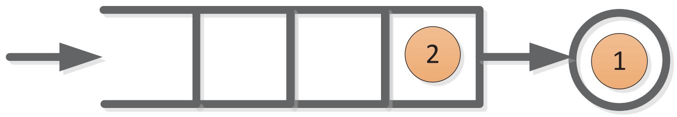
(c) Scheduling of Event A at time 2



(d) Scheduling of Event G at time 10



(e) Execution of Event A at time 2



(f) Final State of the Queue

Figure 6: Example event sequence.

On the third request update message, shown in Figure 6e, the Generate event is moved to the past event list and shaded gray, and the nearest pending event, Arrive, is colored green to indicate that it is the current event. Since this event was scheduled with a time increment of 0 time units, it has a time stamp of t = 2. Because the server is busy, the Arrive event has only a single action, the update of state variables; no future events are scheduled. The updated system model following the execution of the Arrive event is shown in Figure 6f. The system state variable values now are Q(2) = 1 and B(2) = 1.

As shown in this simple example, the visualization tool clearly captures the sequence of event actions that occur during the simulation of the model. At this increased level of resolution, it is much easier to observe and understand the behavior of the simulation.

6 INITIAL RESULTS

The visualization tool is currently being fielded as an academic tool (spring 2017). It is being introduced in an undergraduate discrete event simulation course using the Excel interface with spreadsheet simulations and both graduate and undergraduate simulation software design courses using the C++ interface. This presents the opportunity to assess the benefits of this extra layer of modeling in the learning process. Testing the visualization tool with students learning about discrete event simulation will help assess the benefit of using the tool to better acquire an understanding of event execution and scheduling. Some important questions include:

- Are students able to analyze the system and explain characteristics of the event model?
- Are students able to solve different problems in simulation development through use of the visualization?
- How does using the visualization tool compare to other methods for discrete event simulation, such as hand simulation or debug output?
- How will students interpret or respond to the visualization?

Initial feedback has been gathered from the undergraduate simulation software design course on the use of the interface. Note that it is early in the semester and these results are based solely on the students second assignment. Students were assigned to develop a C++ simulation for the worker interference model example presented as an event graph in Buss (1996). Students were provided Buss's paper and the netlist for the event graph to initialize the simulation, and required to develop software for the example and test their implementation using the event graph visualizer. While a small sample size, feedback was very positive. Students found it easy to interface their software with the tool. They also found the event graph visual very informative, with the event list visual slightly less informative. They were somewhat less enthusiastic about the console output, several not even taking advantage of this feature. It is believed that better instructions and more practice will improve this result.

We also are beginning to work with faculty having expertise in educational assessment to properly construct the test instruments and to more formally gather data and analyze the results.

7 FURTHER RESEARCH

Event graphs have two features not currently represented in the tool. The first is cancellation edges introduced by Schruben (1983). Cancellation edges allow one event to cancel another event. Should multiple events of the same type exist in the event list, Schruben presents a set of rules to identify the appropriate event to cancel. The second feature is the ability to parameterize edges presented in Schruben (1995). An edge is marked by an attribute which is passed to the scheduled event on execution. The existing tool is capable of this feature by outputting the parameter to the console for each scheduling action. On execution of the event, the parameter can again be displayed in the console. However, a more visual approach will be considered.

The visualization tool currently supports communication with simulations created in Microsoft Excel or using C++. These two environments were identified and integrated for their use in educating modeling and simulation students. However, the tool is potentially useful for the development and debugging of simulations. To this end, it is envisioned that the tool could integrate with existing simulation software tools and that the scalability of the tool needs consideration. Integration with the simulation software, Arena, has been identified as a near term goal. But fielding the tool for general use requires the tool to accommodate much larger models than those typically utilized in an academic environment. The initial goal was to handle event graph examples found in the literature, of which the largest is found in Sargent (1988) where he develops a flexible manufacturing model consisting of 14 events. However, an extensible approach is desired. Schruben (1995) presents a hierarchical event graph approach where nodes in the graph could represent a subgraph, but does not propose a visual approach. Future work will consider using a format similar to Harel Statecharts (Harel, 1987), which are now common in the Unified Modeling Language (UML), to provide a visual of a nested event graph. In addition, Buss (1996) discusses using parameterized edges as an approach to representing complex models with simpler components, a capability the current tool provides in an initial form using the console output.

Lastly, the importance of the event graph in the visualization tool matching the event model implemented in the simulation has been noted. If the event graph is developed in the design process, and the simulation is implemented to the specifications of the event graph, this process is straightforward. Support in mapping standard model representations to event graphs would facilitate this process, such as the work found in Schruben and Yucesan (1994). However, this is not always possible, especially in the case of utilizing simulation tools or libraries where parts of the event model are hidden from the developer. For instance, it may not be clear which of the event graphs from Figures 2 and 3 are implemented in a G/G/1 block used in a simulation tool. Therefore, future research could include reverse engineering the event graph from an existing simulation. This capability would provide great insight into the simulation implementation, making the verification and validation process more transparent. This effort could involve a software analyzer to statically derive the event graph from a software implementation. Alternately, an experimental process could be developed to expose the underlying event behavior.

ACKNOWLEDGEMENTS

This work was funded through Old Dominion University's Undergraduate Research Program.

REFERENCES

Bergen-Hill, T., and E. Page, "Out-of-Order Execution and Structural Equivalence of Simulation Models," in *Proceedings of the 2010 Winter Simulation Conference*, Dec. 2010.

Buss, A., "Modeling with Event Graphs," in *Proceedings of the 1996 Winter Simulation Conference*, Dec. 1996.

Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231-274, 1987.

IBM, "IBM Rational Rose Modeler," http://www-03.ibm.com/software/products/en/rosemod, accessed Dec. 12, 2016.

Karagkasidis, A., "Developing GUI Applications: Architectural Patterns Revisited," in *13th Annual European Conference on Pattern Languages of Programming*, July 2008.

Sargent, R., "Event Graph Modelling for Simulation with an Application to Flexible Manufacturing Systems," *Management Sciences*, vol. 34, no. 10, 1988.

Schriber, T., D. Brunner, and J. Smith, " Inside Discrete-Event Simulation Software: How it Works and Why it Matters," in *Proceedings of the 2016 Winter Simulation Conference*, Dec. 2016.

Schruben, L., "Simulation modeling with event graphs," *Communications of the ACM*, vol. 26, no. 11, pp. 957–963, 1983.

Schruben, L., "SIGMA - A Graphical Approach to Teaching Simulation," *Journal of Computing in Higher Education*, vol. 4, no. 1, pp. 27-37, 1992.

Schruben, L., and E. Yucesan, "Transforming Petri Nets into Event Graph Models," in *Proceedings of the 1994 Winter Simulation Conference*, Dec. 1994.

Schruben, L., “Building Reusable Simulators using Hierarchical Event Graphs,” in *Proceedings of the 1995 Winter Simulation Conference*, Dec. 1995.

SigmaWiki, “Sigma: Event Graph Modeling,” http://sigmawiki.com/sigma/index.php?title= Main_Page, accessed Dec. 13, 2016.

Suranauwarat, S., "A CPU scheduling algorithm simulator," in 2*007 37th Annual Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports*, Oct 2007.

Yucesan, E. and L. Schruben, “Structural and Behavioral Equivalence of Simulation Models,” *ACM Transactions on Modeling and Computer Simulation*”, vol. 2, no. 1, pp 82-103, 1992.

AUTHOR BIOGRAPHIES

SAMANTHA C. COLLINS is currently enrolled in the linked Bachelor of Science in Modeling and Simulation Engineering to Master of Science in Modeling and Simulation degree program at Old Dominion University. Her email address is scoll003@odu.edu.

NATHAN D. GONDA is an undergraduate student majoring in Modeling and Simulation Engineering with a minor in Computer Science at Old Dominion University. He has earned an Associate in Computer Science from Paul D. Camp Community College and has experience in computer programming for about 5 years. His research interests are in modeling and simulation visualization and computer game design. His email address is ngond002@odu.edu.

LEE C. DUMALIANG is a current senior at Old Dominion University pursuing a major in Modeling and Simulation Engineering and minors in Computer Science and Engineering Management. He has several years of experience with software development in the defense industry. His email address is lduma002@odu.edu.

JAMES F. LEATHRUM, JR. is an Associate Professor in the Department of Modeling, Simulation and Visualization Engineering at Old Dominion University. He holds a Ph.D. in Electrical Engineering from Duke University. His research interests include simulation software design, distributed simulation, and simulation education. His email address is jleathru@odu.edu.

ROLAND R. MIELKE is a University Professor in the Department of Modeling, Simulation and Visualization Engineering at Old Dominion University. He holds a Ph.D. in Electrical and Computer Engineering from the University of Wisconsin-Madison. His research interests include systems theory, theory and application of simulation, and simulation education. His email address is rmielke@odu.edu.