The Effect of Modeling Simultaneous Events on Simulation Results

John M. Carboni, Douglas D. Hodson, John O. Miller, and Jeremy R. Millar
Air Force Institute of Technology
Wright-Patterson AFB, OH

Abstract- Explored here is the method that governs the prioritizing process for simultaneous events in relation to simulation results for discrete-event simulations. Specifically, it contrasts typical discrete-event simulation (DES) execution algorithms with how events are selected and ordered by the discrete-event system specification (DEVS) formalism. The motivation for this research stems from a desire to understand how the selection of events affects simulation output (i.e., response). As a particular use case, we briefly investigate the processing of simultaneous events by the Advanced Framework for Simulation, Integration and Modeling (AFSIM), a military discrete-event combat modeling and simulation package. To facilitate the building of classic DEVS-based models, the python software package PythonPDEVS is used. Initial results indicate that the explicit modeling of how simultaneous events are selected as promoted by the DEVS formalism plays a significant role on simulation results.

Keywords- DEVS, PythonPDEVS, AFSIM, Simultaneous Event

I. INTRODUCTION
The Advanced Framework for Simulation, Integration and Modeling (AFSIM) is a military discrete-event-based combat modeling and simulation package. AFSIM, like other DES packages, sorts scheduled events using a priority queue, but also like other packages, prioritizes the execution of events (scheduled at the same time) in an arbitrary manner. In other words, simultaneous events are scheduled for execution based on their order of insertion (as determined by software execution flow). We believe this ordering and execution influences simulation output. Additionally, AFSIM (like other discrete-event-based simulations) provides no explicit method to model or determine the processing order of scheduled simultaneous events. We believe the execution order associated with simultaneous events is an important aspect to any simulation; knowing how a model behaves when simultaneous events are scheduled allows one to know their model’s bias and adjust accordingly when analyzing results. Two or more models may have events scheduled to occur at the same time but there is always an order to which model executes first in a computer simulation. This research seeks to understand the effect the execution order on simulation results - specifically, it illuminates this issue as a modeling concern.

The Discrete Event System Specification (DEVS) formalism, which lays out well-defined parameters for how to create models, model systems, and handle simultaneous events, will be used to evaluate this issue. There are many variations of the DEVS formalism; this research considers the classic formalism. In DEVS the atomic model is the fundamental building block for defining a system behavior. Atomic models are connected to create so-called coupled models which lead to assembling a more complex hierarchy. DEVS defines a “select function” which provides an avenue to explicitly define how simultaneous events are to be processed. This research leverages a python-based DEVS software package called PythonPDEVS, which allows a programmer to explicitly define the “select function”. Because DEVS forces a modeler to explicitly define this aspect of behavior, more insight is gained in understanding simulation outputs.

II. BACKGROUND
AFSIM- AFSIM is a combat modeling simulation framework that leverages DES to process events of interest. “AFSIM was developed to address analysis capability shortcomings in existing legacy simulation environments as well as to provide an environment built with more modern programming paradigms in mind” [1]. AFSIM was originally called Analytic Framework for Network-Enabled Systems (AFNES) and was developed by Boeing. In February of 2013 Boeing, under contract, delivered AFNES to the Air Force with unlimited government rights. The Air Force Research Lab “rebranded AFNES as AFSIM and has begun to distribute AFSIM within the Air Force and DoD, including DoD contractors” [1]. AFSIM is used by the Air force to simulate the warfighter in various scenarios. It is a government developed C++ framework used for “constructing engagement and mission-level analytic simulations for the operations Analysis community, as well as virtual experimentation” [1]. AFSIM can simulate conditions from sea-level to space and evaluate the efficiency of a military mission in those domains. AFSIM can simulate a myriad of military systems and weapons using its agent-based software. To name a few, it can model “weapon kinematics, sensor systems, electronic warfare systems, communication networks, advanced
tracking, correlation, and fusion algorithms, and automated tactics and battle management software” [1]. AFSIM has three main parts; the framework itself, the Integrated Development Environment (IDE), and the Visualization Environment for Scenario, Preparation and Analysis (VESPA).

Figure 1: Comment in AFSIM Header File

A central question of concern in this research is how the order of processing simultaneous events affects simulation results. Since AFSIM provided the motivation for this research, a quick investigation of how it handles this conflict was conducted. As seen in Figure 1, there is no criteria on which events are sorted other than time. It leverages a priority queue ordered by simulation time. An undetermined priority governs the handling of simultaneous events in AFSIM. This comment led to the inspiration that motivated this research. It should be noted, it is unknown as to whether AFSIM has simultaneous events happen often or at all. This research utilizes a contrived example in which simultaneous events happen often.

Classic DEVS- DEVS has been around for over forty years and is a widely accepted and applied framework in the modeling and simulation community. The DEVS formalism was developed in the 1970s by Bernard P. Zeigler. “As a mathematical basis for discrete event modeling, DEVS provides not only a formal representation of discrete event dynamic systems that is independent of any computer realization, but also a guideline for how to build abstract DEVS simulation engines to simulate the models” [2]. This research uses the classic DEVS formalism. Classic DEVS is the original version of DEVS, supporting sequential DES modeling and simulation. “Main advantages of DEVS compared to other discrete event formalisms are its rigorous formal definition, and its support for modular composition” [3].

In classic DEVS, the atomic model is “the indivisible building block of a model” [3]. The atomic model is described as an 8-tuple:

\[ M = (X, Y, S, q_{init}, \delta_{ext}, \delta_{int}, \lambda) \]

- \( X \) is the set of input events
- \( Y \) is the set of output events
- \( S \) is the set of model states
- \( q_{init} \) is the initial state of the model
- \( \delta_{ext} \) is the external transition function
- \( \delta_{int} \) is the internal transition function
- \( \lambda \) is the output function

An input event in DEVS is something that triggers the model to transition from one valid state to another. A transition is what defines the movement between states. Internal transitions are due to the passage of time in the simulation as given by the time advance function. External transitions are caused by an input from another model. “When there is no external event, the time interval the model stays on its current state is determined by applying the \( \delta_{ext} \) function to the current state. And the next state of the model is determined by \( \delta_{int}(s) \), where \( s \) is the current state” [2]. Only internal transitions trigger a model’s output function, “the output function is defined on the state, and deterministically returns an event (or no event), the event is generated before the new state is reached. This means that instead of the new state, the output function still uses the old state (i.e., the one that is being left). For this reason, the output function needs to be invoked right before the internal transition function” [3].

Multiple atomic models can be coupled together to create a coupled model. Atomic models are coupled together via their input/output ports (shown below as \( C_{xx}, C_{yx}, C_{yy} \)). A coupled model is also described as a tuple:

\[ M = (X, Y, D, \{M_i\}, C_{xx}, C_{yx}, C_{yy}) \]

- \( X, Y \) are input/output event sets
- \( D \) is the component model name set
- \( \{M_i\} \) is the set of component models
- \( C_{xx}, C_{yx}, C_{yy} \) are the input, internal, output couplings
- \( Select \) is a tie-breaker function

Coupled models can also be treated as atomic models to create more complex scenarios, this is called closure under coupling. “Coupled models are not distinguishable from atomic models when they are coupled with atomic models. Based on the feature of closure under coupling of DEVS, complex system can be hierarchically constructed by model coupling” [2].
The select function in a coupled model is what is of interest for this research. This is what determines processing order for simultaneous events. “This function takes all conflicting models and returns the one that gets priority over the others. After the execution of that internal transition, and possibly the external transitions that it caused elsewhere, it might be that the set of imminent models has changed” [3].

PythonPDEVS- There are many different software tools that implement the DEVS formalisms: ADEVS, CD++, DEVS-Suite, MS4 Me, PowerDEVS, PythonPDEVS, VLE, and X-S-Y. Each tool supports specific formalisms and has its own features, “these tools have distinct design goals and a specific programming language implementation” [4]. Three of these tools were more suited to this research than the rest; ADEVS, CD++, and PythonPDEVS. Using ADEVS wasn’t an option because it does not have the required select function for simultaneous events (as shown in Figure 2). CD++ is a very popular tool and it supports classic DEVS, but not as well as required. In particular, CD++ does not allow the user to define their own select function, “CD++ and X-S-Y implicitly use a hard-coded select function, such as selecting the first model after alphabetic sorting on model name” [4].

Of the tools supporting classic DEVS, “only PythonPDEVS allows users to define the select function explicitly” [4]. PythonPDEVS supports the functionality required for this research while also providing much documentation, good examples, and accessibility to its creators. It is also, arguably, quicker to learn and use since it utilizes the Python programming language. “Due to its implementation in Python, an interpreted, dynamically typed language, fast prototyping of models becomes possible. Despite its interpretation-based nature, PythonPDEVS attempts to achieve high performance. Both atomic and coupled models are written in Python, making (re)compilation unnecessary”[4]. PythonPDEVS will be the tool used by this research to create examples that test the effect of modeling simultaneous events on simulation results.

III. EXPERIMENT

The experiment objective is to understand how the modeling of simultaneous events can affect the results of a simulation. The experiment will focus on simple examples modeling a traffic system. The first scenario is a rudimentary model containing one observer and two traffic light models. Which light will the observer see change first? The purpose is to provide a very straightforward example of the select function at work. The second scenario is slightly more complex. Models for a traffic light, policeman, and car will interact in expected ways as in real life, but the important part is how they interact at that moment when the light changes from yellow to red and the car crosses the intersection at the same time. Does the car get ticketed by the policeman or succeed in crossing the intersection? How the select function is defined for this scenario will directly impact the results of this experiment.

There are seven different ways to define the select function for scheduling priority to take place: Policeman (P)-Light (L)-Car (C), P-C-L, L-P-C, L-C-P, C-L-P, C-P-L, and a random draw. Expectations are for the variants giving priority to the policeman to be modeling a strictly enforced intersection, one with a cop diligently watching and/or a traffic light camera. This would represent the high end of the spectrum, many calls to the select function and multiple tickets given. Giving scheduling priority to the car would represent the low end of the spectrum, modeling an intersection that is much less strict. The random draw and the light would represent the middle of the spectrum and probably be closer to the average intersection.

Example Model 1: The first model made is a very simple example of the select function at work, there are two traffic light models changing from red to green to yellow in a cycle at the same internal transition times. There is also an observer model that just looks at the lights and outputs which light it sees first. This allows for a very easy definition of the select function and understanding of its effect on the simulation. Defining the select function to prioritize Light_1’s internal transition over Light_2’s means the external transition for the observer will always notice Light_1 first, and vice versa for prioritizing Light_2. We can already see, even in this simple example, the kind of impact the select function has on the simulation’s results. Expanding this example could have the observer behaving differently for seeing Light_2 first as opposed to Light_1.
Example Model 2: The second example is slightly more complex with three models. There is a traffic light, a car driving through the intersection, and a policeman watching the intersection. All internal transition times match up with that of the traffic light. The traffic light initializes in the “red” state for 60 seconds, transitions to its “green” state for 50 seconds, transitions to its “yellow” state for 10 seconds, and the cycle repeats. The car initializes in the “stopped” state for 60 seconds and then has the internal transition to the “continue_through” state for 60 seconds. The policeman initializes in the “alert” state for 60 seconds and then transitions to its “idle” state for 60 seconds. The policeman also has a one second “give_ticket” state that it can externally transition to if it is already in its “alert” state, the traffic light is in its “red” state, and the car is in its “continue_through” state. Every transition is based off the traffic light to focus on the changes that result with the differing select function definitions. At every transition three models are trying to execute at the same time, results of this model depend even more on the definition of the select function. If there is no defined select function then the priority of simultaneous events is based on the alphabetical order of model names (i.e “car”, “policeman”, then “trafficLight”).

Testing: Testing was done based on the different ways to define the select function. The random definition testing consisted of 10,000 trials of 1,000 second simulations to collect a large amount of data. All the other defined select function tests are deterministic and need only be run once. Figure 3 illustrates data collection.

![Figure 3: Test Matrix for Select Function Calls](image)

Each trial is one run of the simulation, the random definition will run the simulation 10,000 times. I expect the number of times the select function is called to be predictable; policeman phases will have the most, car phases will have the least, and light/random phases will be somewhere in the middle. The non-random select function definitions are deterministic and will only be run once.

![Figure 4: Test Matrix for Model State Occurrences](image)

Figure 4 displays how data collection will be done for the number of times each state occurs in each definition of the select function (average number for random definition). This data will help further determine the effect explicit model priority has on simulation results.

IV. RESULTS

The predictions were almost correct. The expected results for the car and the light were switched, but the random and policeman results were as expected. Figures 5-8 display the collected data for each select function definition. We can see that results differed significantly depending on which model had highest priority, but the second highest priority doesn’t seem to have much of an effect. Perhaps most interesting are the results for the random definition and how it compares to the other, explicitly defined, phases.

![Figure 5: Calls to the Select Function](image)

![Figure 6: Statistics for Random Definition (Select Function Calls)](image)
Figure 7: Model State Occurrences

<table>
<thead>
<tr>
<th>(Select Definition)</th>
<th>Alert</th>
<th>Idle</th>
<th>Red</th>
<th>Green</th>
<th>Yellow</th>
<th>Continue</th>
<th>Stopped</th>
<th>Ticket</th>
</tr>
</thead>
<tbody>
<tr>
<td>P-L-C</td>
<td>33</td>
<td>24</td>
<td>8</td>
<td>8</td>
<td>40</td>
<td>17</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P-C-L</td>
<td>41</td>
<td>32</td>
<td>9</td>
<td>8</td>
<td>40</td>
<td>23</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>L-P-C</td>
<td>17</td>
<td>16</td>
<td>9</td>
<td>8</td>
<td>24</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>L-C-P</td>
<td>17</td>
<td>16</td>
<td>9</td>
<td>8</td>
<td>24</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C-L-P</td>
<td>9</td>
<td>32</td>
<td>9</td>
<td>8</td>
<td>24</td>
<td>25</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>C-P-L</td>
<td>28</td>
<td>32</td>
<td>9</td>
<td>8</td>
<td>24</td>
<td>41</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

Random 25.688 15.2413 9 8 29.254 22.2778 2.6188

Figure 8: Statistics for Random Definition (State Occurrences)

Policeman- When the policeman has the highest priority, it results in more simultaneous events (48 select function calls) than any of the other definitions tested. This makes sense as how our models were defined, the policeman model’s internal transition doesn’t cause external transitions in the other models. The policeman switches states from “alert” to “idle”, and vice versa, acting as an observer. Due to this, the simulation must play out the next simultaneous event between the other two models every single time.

The second highest priority in these phases cause a slight difference in the actual trace results of the simulation. If the traffic light model has the second highest priority then the simulation results in a smooth intersection, no tickets given. However, if instead the car model is given the second highest priority then the resulting trace contains more data, but still no tickets given. With highest priority, the policeman model changes state to “idle” first before the traffic light model changes state to “green”. When the car model has the second highest priority it can run the red light every time because its state changes to “continue_through” before the traffic light transitions to “green”. There is more data in the trace because the traffic light model has the most influence over the other two models. Every state change in the traffic light causes external transitions in the other two models, so when it finally changes state with its last priority it causes external transitions that output redundant data. Figure 7 shows the increase in state occurrences between the P-L-C and P-C-L scenarios.

Traffic Light- The traffic light resulted in the fewest simultaneous events when it had highest priority, only 24 calls to the select function. This is because the traffic light model is the main influencer in this simulation. Every time the traffic light changes state it will cause an external transition in the other two models and eliminate their next internal transition. This also eliminates the next simultaneous event that would have happened between the car and policeman models, the light has already dictated their next state. With the priority of the light diffusing any further scheduling conflicts, it follows that the select function would be utilized less.

These scenarios are the only ones where the second highest priority doesn’t matter. As mentioned earlier, the traffic light model transitions first every time and dictates the states of the other two models. That leaves no further priority required, the other models will only act in external transitions via the traffic light. We see this further exemplified in the trace results for both the L-P-C and L-C-L scenarios, Figure 7 shows identical results for state occurrences.

Car- In the explicit definitions with highest priority given to the car model, the select function was called 40 times. This is the closest average to that of the random scenario. The car model, like the policeman model, doesn’t have much influence. The only external transition the car model causes is in the policeman model. When the car is in state “continue_through” and the traffic light is in state “red”, the policeman transitions to a “give_ticket” state. Giving the car priority results in the most tickets issued. The policeman is still in its “alert” state and the traffic light is still in its “red” state when the car changes state from “stopped” to “continue_through”. The policeman model’s external transition occurs when this happens and gives the car a ticket. Like in the traffic light priority definitions, this external event will eliminate some subsequent simultaneous events from occurring (hence 40 calls) but not all.

The C-P-L and C-L-P scenarios are the only ones that yield any tickets given by the policeman model. The trace results for both definitions show eight tickets given over the course of the 1,000 second simulation. Much like when the policeman had highest priority, there is only a slight difference caused by the second highest priority. The C-P-L trace has more data due to the same reason as the P-C-L trace. When the traffic light model has last priority, it will cause external transitions in the other models that, at that point, are redundant.

Random- For 10,000 trials of the 1,000 second simulation with random priority, results yielded an average of about 37.3 calls of the select function with a range from 29 to 45 calls. The random priority never reached as high as 48 calls nor as low as 24 calls like we saw on the explicit definitions. Giving priority at random each time a simultaneous event occurs.

<table>
<thead>
<tr>
<th>Random</th>
<th>give_ticket</th>
<th>stopped</th>
<th>yellow</th>
<th>red</th>
<th>idle</th>
<th>green</th>
<th>continue_through</th>
<th>alert</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>2.6818</td>
<td>22.2778</td>
<td>8</td>
<td>9</td>
<td>25.2813</td>
<td>8</td>
<td>29.354</td>
<td>23.668</td>
</tr>
<tr>
<td>max</td>
<td>8</td>
<td>35</td>
<td>9</td>
<td>16</td>
<td>18</td>
<td>37</td>
<td>8</td>
<td>40</td>
</tr>
<tr>
<td>min</td>
<td>0</td>
<td>11</td>
<td>8</td>
<td>9</td>
<td>20</td>
<td>16</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>stdev</td>
<td>1.33054535</td>
<td>3.215467</td>
<td>0</td>
<td>0</td>
<td>2.837826</td>
<td>0</td>
<td>2.75406681</td>
<td>1.88765</td>
</tr>
<tr>
<td>var</td>
<td>1.77038476</td>
<td>10.3388</td>
<td>0</td>
<td>0</td>
<td>8.16717</td>
<td>0</td>
<td>7.584884</td>
<td>8.33805</td>
</tr>
</tbody>
</table>

Figure 7: Model State Occurrences

Figure 8: Statistics for Random Definition (State Occurrences)
provides a wider range of possibilities in the simulation, but we can see it doesn’t account for the scenarios that we explicitly defined. I also noticed that the average of the other three highest priorities (48, 24, 40) also comes out to 37.3 calls. As expected, the random priority results came out to be the average of the other results.

The difference in traces for the random priority show a range of values for each state that excludes some values from the explicit definitions. For example, in Figure 7 we can see the “stopped” state has occurrences as high as 41 and as low as 9, but in Figure 8 we see the random results for “stopped” range from 11 to 35. Only the “red”, “green”, and “yellow” states occur at a constant rate throughout the random trials. These results aren’t surprising, no other model has any influence over the traffic light model’s transitions. Traffic light states will always occur no matter the priority. Unlike the traffic light model, the policeman and car models’ states show much greater variation. The average number of occurrences for their states for the random definition falls in the middle range of the explicit scenario data, except for the “give_ticket” state. The random average for “give_ticket” is 2.6818 with a range from 0 to 8. This is the only state with an average that doesn’t land somewhere near the middle of the range. However, 2.6818 is very close to a third of eight and tickets are only given when the car model has priority. Having the average number of occurrences for this state land around the one third mark in the range makes sense since only one of three models contributes to the occurrences of this state.

V. CONCLUSION

The examples in this paper have shown that the handling of simultaneous events in modeling and simulation can make a difference in one’s results. Proper consideration of this concept of modeling and simulation can be impactful when trouble shooting ones models, gathering data, and analyzing data.

Coding the models and simulation conditions is no easy task. Often is the case that the simulation isn’t acting as intended and one must figure out why and fix it. Just the knowledge alone of how the simulation handles simultaneous events is a boon to trouble shooting. It makes it easier to walk through what is happening in the simulation and pinpoint the spot errors begin to occur.

Testing a simulation can take an extraordinary amount of computing time, especially if one is testing for every possible scenario. Accounting for simultaneous events by explicitly defining model priority can shorten testing time and reduce noise in results by eliminating the gathering of redundant/useless data.

There is also a benefit to the analysis of data when simultaneous event handling is properly accounted for. It adds an extra layer in results analysis between “what happened?” and “what does it mean?”. Now, there is also “why it happened that way?” which adds extra insight to the other layers for a more complete analysis.

We know this is true for DEVs, as it was the DES used in this research, but it would be interesting to repeat this experiment with other DES and see if the results are similar.

VI. FUTURE WORK

A great continuation of this research would come in the form of a similar experiment, but in Parallel DEVs using the PythonPDEVs tool. Like the select function in classic DEVs, Parallel DEVs has the confluence function. The confluence function basically serves the same purpose as the select function, but it is not restricted to internal transition priority between models. It also determines priority for external transitions. It would be interesting to replicate this research using Parallel DEVs and see how results change [5].

Other future work would replicate this research using other DES tools. It would be interesting to see if other DES tools behaved similarly according to their respective versions of the select function, if they even have one. A survey paper of other DES tools regarding which have implicit handling of simultaneous events, which let you explicitly define model priority, and which neglect it altogether would be good future work.

VII. REFERENCES


