Arena vs The World: A Comparison of Simulation Language

Course Project for ISYE 6644

Project Group #214: Inah Canlapan

1. Abstract

The purpose of this paper is to compare and contrast Arena to SimPy, an open source simulation framework based in Python. In the literature review, comparisons of different simulation programs and languages are made by Leeming, Göbel, Joschko, Koors and Page. Included in these reviews are GPSS, ESCL, Simula 67, DESMO-J and other Java based simulation libraries. Arena and SimPy will reviewed at a high-level and will be used to simulate a passenger's journey through a busy airport where they must pass through two different types of scanners. While Arena is superior to SimPy when it comes to visualizing a process, learning how to use the GUI based program may be more challenging for Python based programmers who are used to creating their own programs. Both frameworks have similar functionality but the best framework of choice is ultimately up to the programmer.

2. Background/Literature Review

In 1981, Leeming compared three simulations languages: GPSS (General Purpose Simulation System), ESCL (Extended Control and Simulation Languages) and Simula 67. They simulated the same sample problem in each of the languages and evaluated them based on the same 7 features: view of the problem taken by the language, fundamental concepts, scheduling procedures, I/O facilities, random number provision, program structure, debugging and ease of learning. A comparison of computational resources used by the each of the programs was also included (Leeming, 1981).

In Leeming's description, GPSS seems similar to Arena. The programmer creates flowcharts to create the simulation model and uses ready-to-use components, also called blocks (ex – GENERATE, TERMINATE, QUEUE), to create actions where one can the specify parameters that govern its behaviour. While GPSS was the cheapest to use in terms of computational resources, is well documented and was relatively easy to learn, there are no other applications of GPSS outside of simulation. A GPSS programmer, similar to an Arena programmer, will not be able to use their skills in any other medium (Leeming, 1981).

Simula 67 is more flexible compared to GPSS as it can be used for any application, not just simulation. The programmer must breakdown their problem into its fundamental units, or objects as they are referred to in Simula, and can group them into classes. These classes of objects can then be used on their own or as building blocks in a larger problem (Leeming, 1981). This approach to problem solving is similar how one would use libraries in open source simulation languages based in Python to create a simulation model.

Göbel, Joschko, Koors and Page focused their research on DESMO-J, a Java based open source simulation library. They compared DESMO-J to seven other Java based simulation frameworks that a programmer could choose from like DSOL, J-Sim and JSL to name a few. The comparison was based on 6 features: the support of event and process modelling, availability of 2D or 3D visualization of model behaviour, number of random distributions available, availability of examples and tutorials and commercial use. DESMO-J is superior to all other frameworks on all features – it allows for both event and

process modelling, supports 2D and 3D visualization, has very detailed tutorials on its website and is used in commercial settings (Göbel, Joschko, Koors and Page, 2013).

3. Main Findings

3.1 Arena

Arena is a discrete event simulation software owned by Rockwell Automation that is based on the SIMAN simulation language (Li, Keyser and Han, n.d.). It is a commercial software used in various industries that include health care, retail, supply chain, logistics and mining to name a few (Discrete Event Modeling | Arena Simulation Software, n.d.). There are also simpler versions of the software available for students and academic research (Academic Offerings | Arena Simulation Software, n.d.). Arena can model complex business processes and provide statistical analysis, performance metrics and dashboards that will allow decision makers to assess the impact of system or process changes without disrupting the current systems and ultimately, increase profitability through overall improved operations. (Discrete Event Modeling | Arena Simulation Software, n.d.).

In Arena, a programmer can build models using a large library of predefined building blocks, called modules, and connect them in a flowchart (See Figure 1). It is only compatible with the Windows operating system and will be more intuitive to use if the programmer is familiar with Microsoft's suite of products. Within each of the modules, the programmer can define statistical distributions, assign attributes, create resources and dispose of them at the end of the process.

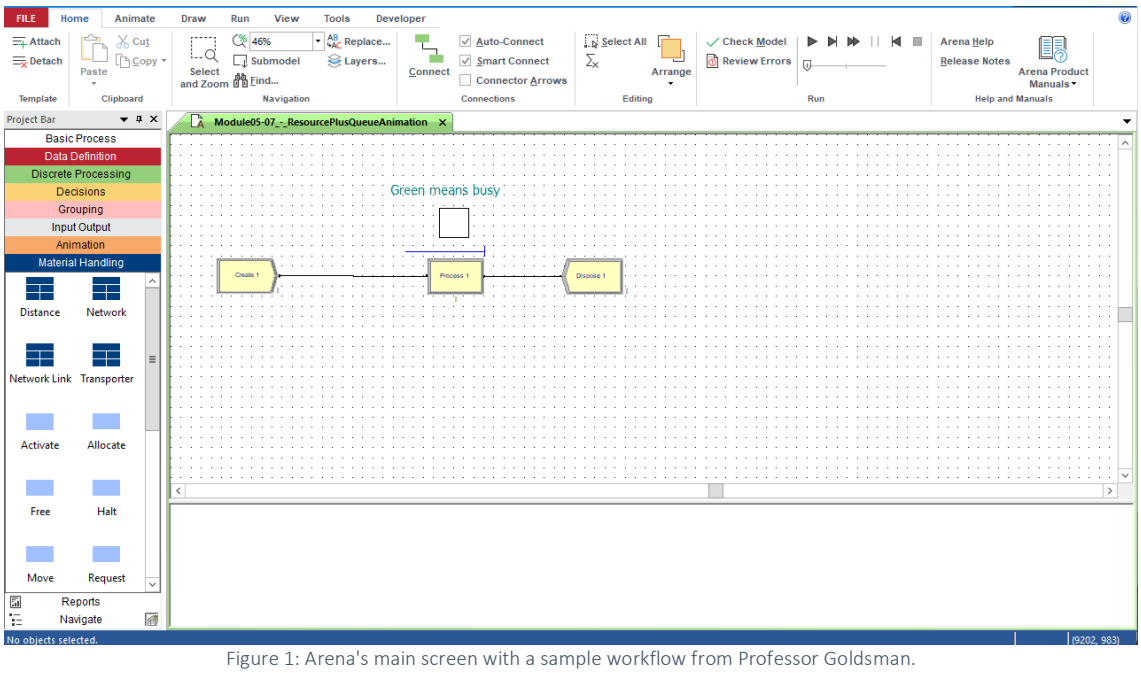


Figure 1: Arena's main screen with a sample workflow from Professor Goldsman.

3.2 SimPy

SimPy is a discrete event simulation framework based in Python that was first released in 2002. It was originally based on ideas from Simula and Simscript, but uses Python. As such, it is an open source library free to use on all operating systems that can run Python code. The first iteration was created by Klaus Müller and Tony Vignaux. SimPy 2, which includes the integration of an object-oriented API into the existing API allowing backwards compatibility, was primarily developed by Stefan Scherfke and Ontje Lünsdorf (Overview — SimPy, n.d.).

To use the package, the user simply needs to install it in their IDE of choice then import it into their code. Once the package is imported, the programmer can define events and schedule them at any given simulation time. They can also use Python generator functions to model simulation components like customers, vehicles and agents. Once a simulation model or environment is created, it can be run in real time or step-by-step through the event (Overview — SimPy, n.d.). Robust documentation for SimPy exists online that includes topical guides and examples that demonstrate how to use SimPy's features.

```
>>> import simpy
>>>
>>> def clock(env, name, tick):
...     while True:
...         print(name, env.now)
...         yield env.timeout(tick)
...
>>> env = simpy.Environment()
>>> env.process(clock(env, 'fast', 0.5))
<Process(clock) object at 0x...>
>>> env.process(clock(env, 'slow', 1))
<Process(clock) object at 0x...>
>>> env.run(until=2)
fast 0
slow 0
fast 0.5
slow 1
fast 1.0
fast 1.5
```

Figure 2: A simple SimPy example that simulates two clocks ticking at different time intervals. (Overview — SimPy, n.d.)

3.3 Comparing Solutions to the Same Problem

In this example from a previous class (ISYE 6501), an airport security system at a busy airport is simulated. Passengers arrive according to a Poisson distribution with $\lambda = 50$ passengers per minute. Servers in the ID & boarding pass check queue have an exponential service time with mean rate $\mu = 0.75$ minutes. Passengers are the assigned to the shortest of the several personal check queues where service time is uniformly distributed between 0.5 minutes and 1 minute.

The goal of the original problem was to vary the number of servers in the boarding check queue and the personal check queue so that the average wait times are below 15 minutes. In the included solutions, the number of document check queues and the number of personal check queues are set to 35 for demonstration purposes. The solutions for both SimPy and Arena that are included were created by the author.

SimPy Solution

The first step in any Python program is to import packages and set the appropriate parameters. These parameters will be passed on to generator functions later on in the program.

```
import simpy
import random
import pandas as pd
import numpy as np
pd.set_option('display.max_columns',10)

num_runs = 25 # Repeat simulation 25 times with the same parameters to get average
num_checkers = 35 # Number of document check queue servers
num_scanners = 35 # Number of personal check queue servers

arrival_rate = 50 # Passenger arrival rate
boarding_pass_mean = 0.75 # Document check queue service time
minScan = 0.5 # Personal check queue minimum service t››ime
maxScan = 1.0 # Personal check queue maximum service time
sim_time = 360 # Each run will be 6 hours long
passenger_list = {}
res = {}
```

The next step in the code is to create the Airport environment and create the resources using the simpy.Resource class. The distributions of the boarding queue servers and the personal scanners are set using the appropriate distributions from the random package and the parameters defined in the previous step.

```
# Creating the Airport environment
class Airport(object):

    def __init__(self, env):
        self.env = env
        self.boarding_checker = simpy.Resource(env, num_checkers)
        self.personal_scanner = []  # Set of scanners
        for i in range(num_scanners):
            self.personal_scanner.append(simpy.Resource(env, 1))

    def boarding_check(self, passenger):
        rand_arrival = random.expovariate(1.0 /boarding_pass_mean)
        yield self.env.timeout(rand_arrival)

    def scan_time(self, passenger):
        rand_scan_time = random.uniform(minScan, maxScan)
        yield self.env.timeout(rand_scan_time)
```

This function models the journey of the passenger through the airport. The .queue method provides the list of passengers waiting at each scanner and the len() function is used to return the number of passengers waiting. Once the scanner with the shortest queue has been identified, that scanner is requested using the .request() function. Timestamps of passenger activity like arrival at airport and entrance and exit from the boarding check and personal scanner are saved in order to calculate wait times later on.

```
def passenger(env, name, s, passenger_list):

    # Time passenger arrives at airport
    passenger_list[name][0] = env.now

    with s.boarding_checker.request() as id_check:
        yield id_check

        # Time passenger enters boarding check
        passenger_check_start = env.now

        yield(env.process(s.boarding_check(name)))

        # Time passenger leaves boarding check
        passenger_check_end = env.now

        # Time passenger spends in boarding check
        passenger_list[name][1] = passenger_check_end - passenger_check_start

    # Checking for the shortest line
    shortest_line = 0
    for i in range(1, num_scanners):
        if (len(s.personal_scanner[i].queue) < len(s.personal_scanner[shortest_line].queue)):
            shortest_line = i

    with s.personal_scanner[shortest_line].request() as scan_request:
        yield scan_request

        # Time passenger enters personal scanner
        passenger_scan_start = env.now
        yield env.process(s.scan_time(name))

        # Time passenger leaves personal scanner
        passenger_scan_end = env.now

        # Time passenger spends in personal scanner
        passenger_list[name][2] = passenger_scan_end - passenger_scan_start

    # Time passenger leaves the checks
    passenger_list[name][3] = env.now
```

This function sets up a brand-new Airport environment and begins to process passengers. It also saves the timestamps collected in the passenger function to passenger_list.

```
def setup(env):
    i = 0
    s = Airport(env)

    while True:
        yield env.timeout(random.expovariate(arrival_rate))
        i+=1
        passenger_list['Passenger ' + str(i)] = [0, 0, 0, 0]
        env.process(passenger(env, 'Passenger %d' % i,
                              s, passenger_list))
```

Now the simulation is ready to begin! For each of the 25 runs, a new Airport environment is created and runs for 6 hours (simulation time, not real time!). Passenger timestamps collected in the passenger_list are converted to a pandas data frame and are used to calculate the average wait time in each run.

```
print("Starting Airport Simulation")

# Run  multiple simulations and calculate wait time
for i in range(num_runs):
    random.seed(i)

    env = simpy.Environment()
    env.process(setup(env))

    env.run(until = sim_time)

    data_time = pd.DataFrame.from_dict(passenger_list, orient = 'index',
                                                       columns = [  'arrival_time',
                                                                    'time_in_boarding_check',
                                                                    'time_in_personal_scanner',
                                                                    'exit_time'])
    # Calculate total time in system
    data_time['total_time'] = data_time['exit_time'] - data_time['arrival_time']

    # Calculate wait time
    data_time['total_wait_time'] =  data_time['total_time'] - data_time['time_in_boarding_check'] - \
                                    data_time['time_in_personal_scanner']
    data_time2 = data_time[data_time['total_time'] > 0]

    # Append avg of this run's passengers wait times to results
    res[i+1] = data_time2['total_wait_time'].mean()

    print("Run " + str(i+1) + " complete")

sim_res = pd.DataFrame.from_dict(res, orient = 'index', columns = ['Avg Wait Time'])
print(sim_res)

# Calculate % of runs where wait time is below 15 mins
success = sim_res[sim_res['Avg Wait Time'] <= 15]
success_rate = success.count()/ sim_res.count()
print("% of runs where wait time is below 15 mins with " + str(num_checkers) +" boarding pass checkers and " \
      + str(num_scanners) + " personal scanners is " + str(round(success_rate[0]*100, 1)) + "%")
print(str(sim_res.mean()))
```

The total run time for Python to execute 25 runs is 56 seconds. To execute one run, it takes about 2.5 seconds.

Arena Solution

Below is an overview of the Arena solution to this problem. It is easy to understand the flow of the process through this visualization and additional features like a clock and counters can be added to enhance the animation.

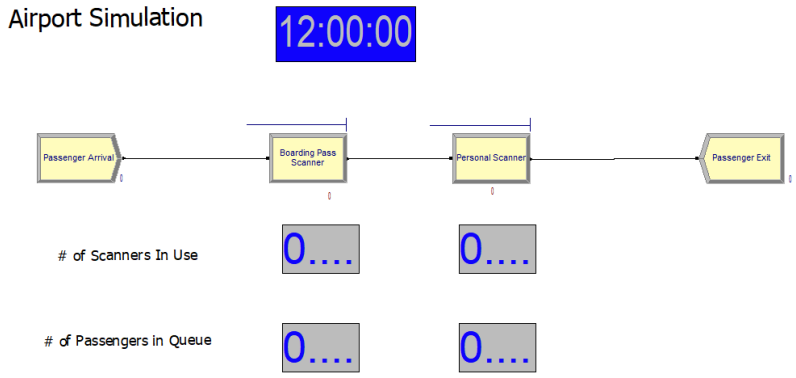
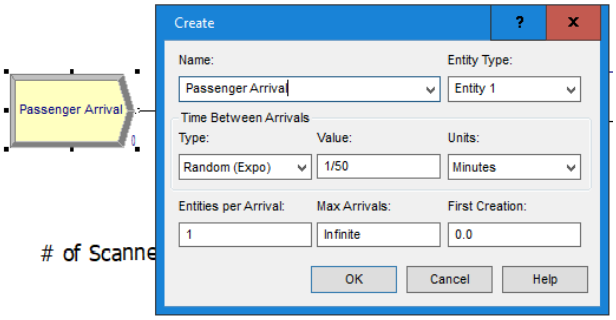
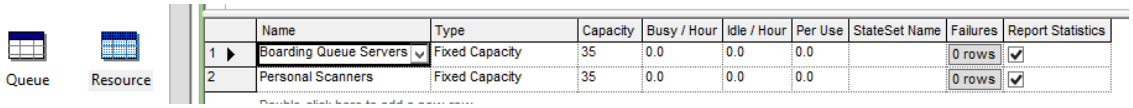


Figure 3: Airport Simulation Overview

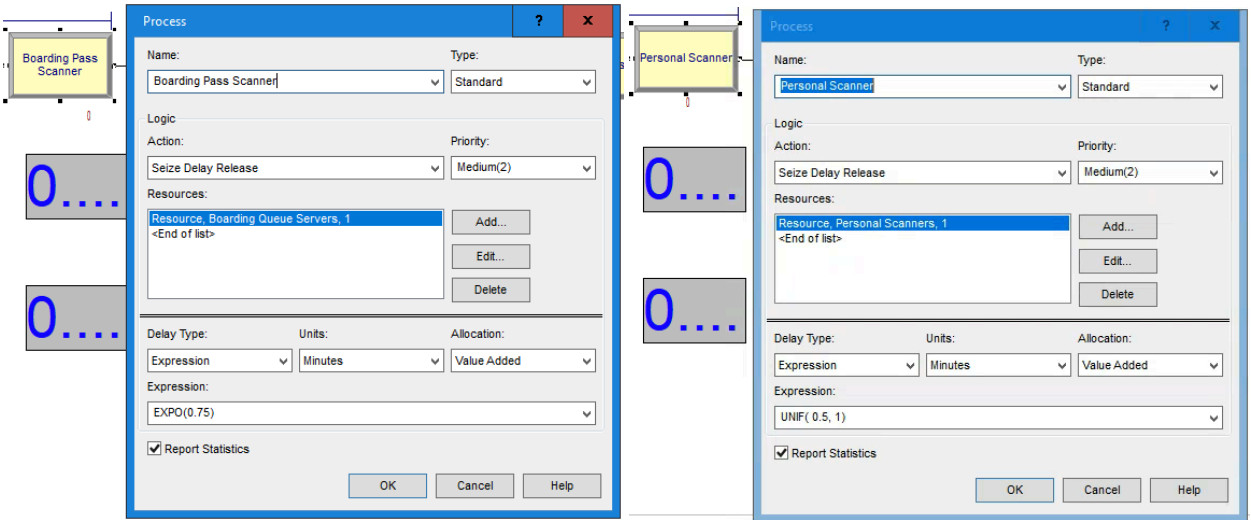
The first step to building this solution was to use the CREATE module to simulate passengers arriving. After dragging the module from the Basic Process template onto the main screen, it can be renamed and the arrival distribution can be set using a schedule, default distributions or a custom expression.



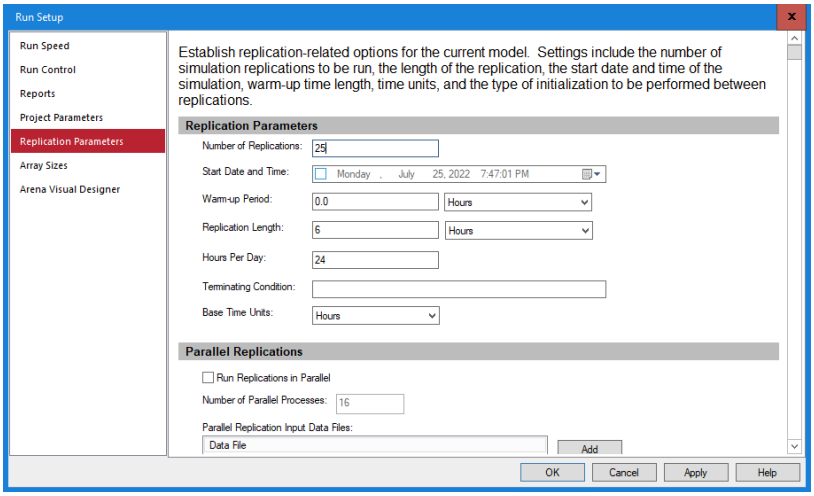
Then, two resources were created using the Resource spreadsheet, one for the boarding pass scanners and another for the personal scanners. The number of each resource available can be set in this spreadsheet.



The next step is to use the PROCESS module to simulate each of the scanners. In each process, one resource from each scanner type will be seized, delayed according to the respective distribution time and released.



The last step in the process is to use the DISPOSE module to finish processing the passengers. Before running the simulation, the replication parameters can be set in Run Setup to match the number of replications (25) and length of run (6 hours) in the SimPy solution.



It takes around 5 seconds to run 25 replications without animations. With animations at the fastest speed, it takes about one minute to complete.

4. Conclusions

There are many different simulation frameworks one can use to model a problem. These programs and languages have existed for a very long time and continue develop, especially those that are based on open source languages like Python, through their developer communities. Arena will be easier to use for programmers that have more experience in GUI based programming and little experience programming in Python. Conversely, SimPy will be easier for native Python programmers to understand and customize since they are used to creating their own programs, functions and/or scripts. One may argue that it is easier to understand what is happening in SimPy because the process is fully exposed; there is no need to click to different windows or spreadsheets in order to understand the components of the process like in Arena. However, when it comes to visualizing and animating the process flow, Arena is the superior tool. Ultimately, the programmer should use the language or program that they are most comfortable and confident in.

References

En.wikipedia.org. n.d. SimPy - Wikipedia. [online] Available at: <https://en.wikipedia.org/wiki/SimPy> [Accessed 20 July 2022].

Göbel, J., Joschko, P., Koors, A. and Page, B., 2013. The Discrete Event Simulation Framework DESMO-J: Review, Comparison To Other Frameworks And Latest Development. [online] pp.100-109. Available at: <https://www.researchgate.net/publication/269230596_The_Discrete_Event_Simulation_Framework_DESMO-J_Review_Comparison_To_Other_Frameworks_And_Latest_Development> [Accessed 16 July 2022].

Leeming, A., 1981. A Comparison of Some Discrete Event Simulation Languages. SIGSIM Simul. Dig., [online] 12(0163-6103), pp.9-16. Available at: <https://doi.org/10.1145/1103225.11032> [Accessed 15 July 2022].

Li, L., Keyser, R. and Han, M., n.d. Systems Simulation. [online] Alg.manifoldapp.org. Available at: <https://alg.manifoldapp.org/projects/systems-simulation> [Accessed 16 July 2022].

Rockwell Automation. n.d. Academic Offerings | Arena Simulation Software. [online] Available at: <https://www.rockwellautomation.com/en-us/products/software/arena-simulation/academic.html> [Accessed 21 July 2022].

Rockwell Automation. n.d. Discrete Event Modeling | Arena Simulation Software. [online] Available at: <https://www.rockwellautomation.com/en-us/products/software/arena-simulation/discrete-event-modeling.html> [Accessed 21 July 2022].

Simpy.readthedocs.io. n.d. Overview — SimPy. [online] Available at: <https://simpy.readthedocs.io/en/latest/index.html> [Accessed 21 July 2022].