Legend

An actor based ground truth simulator

Ryan Stepanek

Legend - Summary

The goal of the Legend project is two-fold:

1) Empower Subject Matter Experts (SMEs) to build their own agent based simulations without the assistance, or with reduced assistance, from a modeling specialist or computer scientist.
    a) Legend achieves this by allowing flat file construction of complicated concepts through intuitive data structures.
2) It must be capable of scaling to meet the needs of the end user.
    a) Using actor based modelling, the Legend system is inherently distributed, allowing it share intensive workloads across different systems and processors.

Legend - Input Files

Each Legend simulation has 6 input files:

1. Sim.config - The config file for the simulation
2. Event List - A list of predetermined events set by the user
3. KML File - A KML file that holds a list of sites and areas of interest
4. Entity File - A file that holds the templates for entity types
5. States File - A file that holds the data for all states of all entities
6. Process File - A file that holds all process data (the graph relationships of states)

*The next planned improvement is to allow splitting of files 2,4,5,6 to load from multiple directories as this will allow for a better user experience

Legend - Input Files (Sim.Config - Part 1)

The Sim.Config file is the first file loaded in every sim run. It loads the following parameters:

start_date = A start date for the simulation. It can be specified down to the second i.e. 2017-05-05 17:21:00

end_date = An end date for the simulation. It can be specified down to the second i.e. 2017-05-08 17:23:01

out_file = The location of the INFO level log file, contains information on the overall system status and progression.

warn_file = The location of the WARN level log file, contains warning which could cause unexpected behavior.

error_file = The location of the ERROR level log file, contains errors which are usually fatal to the simulation.

event_file = The location of the EVENT level log file, contains most of the output of the sim.

kml_location = The location of the scenario kml file.

Legend - Input Files (Sim.Config - Part 2)

The Sim.Config file is the first file loaded in every sim run. It loads the following parameters:

entity_location = The location of the entity template file.

states_location = The location of the states file.

process_location =The location of the process file.

event_location = The location of the user-defined event file.

server_uri = The address of a server to stream EVENT level data. If this is not set, the system will not stream data.

Legend - Input Files (Event File)

Every Legend run must have at least one event (a “spawn” event) defined in an event file. A valid Event File has the following columns (tab delimited):

time - The time at which the event occurs, can be sim time (milliseconds since start) or real world time.

Event - The event to throw and its args.

Number - The number of times to insert an event of this type and args into the queue.

i.e.

```
time      event    number
1         Spawn(entity=test_tank,location=Hangar)          1
3         Spawn(entity=test_tank,location=Runway)          2
2017-05-05 17:23:00     Spawn(entity=test_tank,location=Military Airport) 1
```

*At the moment, only the Spawn event is supported. In the near future, a Destroy event will also be supported.

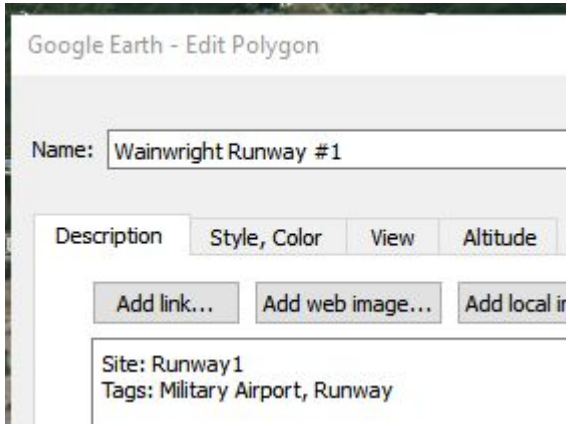
Legend - Input Files (KML File)

A KML file is a convenient way of storing geospatial information. It is highly recommended that you use a tool such as Google Earth to construct your KML file for input into the Legend system. Each entry in the KML should be a polygon which will be converted into a site.

A **site** is an area of interest where we can expect some event may occur i.e. Spawn, GoTo, etc… it is a way of instructing the internal system that a segment on earth, however large or small, is of interest to us and should be indexed and tagged.

In order for Legend to form a valid site from the KML the description property must be populated with at least one of the **site** or **area** parameters, the **tags** parameter is optional.

i.e.



*Currently sites are converted into rectangles equivalent to their bounding box to avoid expensive geometry computations at run time.

Legend - Input Files (Entity File)

The entity file holds all the templates for entities you wish to model in the sim i.e. B-52s, sedans, fire trucks, F-16s, etc… At the moment, an entity consists simply of name of the entity and its default process graph. When the Spawn event is issued, the system will check to make sure that “entity” argument matches the name of an existing entity type. All entity names must be unique.

i.e.
name = Firetruck
process = PutsOutFires

In the future, this will also contain the approximate dimensions and mass of the entity (used for modelling acceleration profiles) as well as any starting resources that it may have by default.

Legend - Input Files (States File)

The states file holds all possible states for all entities. A state represents an entity's state i.e. attributes such as speed for a certain duration. At present, all state names must be unique.

Duration (required) duration can be -1 to indicate an indefinite state, it can be a single value i.e. 4 hours, 3min,14sec, etc.. or it can be a range 1min-3days, 15s-3wks. It has a fairly intuitive parser for time units up to week lengths i.e. 52 weeks is OK, but 1 year or 12 Months is not.

Speed defaults to 0 if not specified. If no units are provided for the the speed, then kph is assumed. Currently supported units are kph,mph, and knots.

Directives are special events only scheduled upon entering a state, at the moment, the only supported user provided directive is GoTo which will make an entity move to either a point or a site with matching tags and then stop when it arrives.

I.e.

Name = State:test2
Duration = 4hrs
Directives = GoTo(location=Runway)
Speed = 10

In the future, additional parameters such as visible/hidden, yields and cost (for modifying resource values), tags (selection of states during for run-time creation of processes), time frequencies for modelling PoL (Pattern of Life) behavior will be supported, and messaging will be supported.

Legend - Input Files (States File)

The process file holds all of the processes within the simulation. Each process must have a unique name. A process is a network graph of states (nodes) and transitions (edges). Note that while syntax for message based transitions exists, it is not yet implemented internally within the system.

A transition is a connection between two states with some probability. When a state has run its duration the process graph is checked to determine which state it should transition to next. At present you are allowed to enter probabilities for a state that do not sum to 1, this may cause unexpected behavior and is not recommended; it may be forbidden in future releases.

A transition may be of the form:
State1 -- 1.00 => State2

Or it may be an entrance transition i.e.

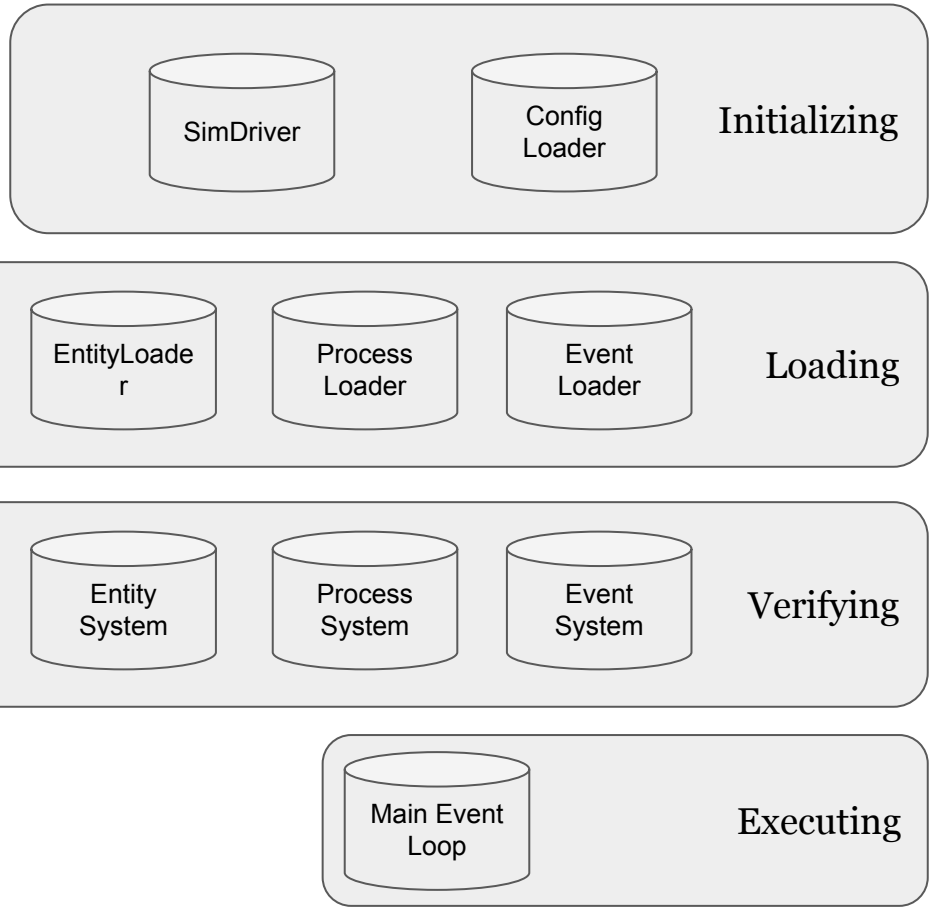
0.80 => State:test
0.20 => State:test2

(a transition upon entering the process graph)

Sample Process File entry

```
Name = Process:test
0.80 => State:test
0.20 => State:test2
State:test -- 1.00  => State:test2
State:test2 -- test_message -- 1.00 => State:test
State:test2 -- test_message1;test_message2 -- 1.00 => State:test2
State:test2 -- 0.30  => State:test
State:test2 -- 0.10  => State:test2
State:test2 -- 0.60  => State:test3
State:test3 -- 0.95  => State:test
State:test3 -- 0.05  => State:test2
```

Legend - Operating Phases



Loading is done in parallel using the config settings. Some minor verification is performed. Data is readied to be turned into sim objects.

Data is taken from the respective loaders and turned into real sim objects. This allows guaranteed verification of data before sim run.

SimEvents are processed and distributed to their respective systems until the SimEnd event it reached.

Note that the SimDriver is the only system to initialize other systems. In the future, the Geometry system will initialize subsystems called localities.

Legend Systems - Part 1

SimDriver - The first system and parent to all the other systems, it coordinates the phase progression of the sim and servers as the host for the address of all systems within the simulation, providing addresses upon request. Lastly, the driver initialize the SimStart and SimEnd times.

Config Loader - A special loader that loads the Sim.Config file, parses it and stores the data. It answers key requests from other systems and it initializes the logging system.

*** Loader** - Loading systems that load specific files in parallel and perform some transformation and cleaning on the data.

Geometry System - Holds all sites loaded in the in the simulation, as well as their tags in a lookup table for quick finds. It handles VelocityChange and GoTo directives, tracking entity destination and time till arrival. In the future it will also handle entity lookup via delegation to the locality subsystems.

Entity System - Maintains the current state of all entities in the simulation, works with Geometry System to handle spawning of entities. Logs events as entity state is updated.

Legend Systems - Part 2

State System - Performs final validation of states. In contrast to initial system design, this system appears largely redundant with most of its expected functionality implemented inside the Process System more intuitively. In future releases the State System will be cannibalized into its loader and into the process system.

Process System - Verifies the processes for entities, handles state changes, alerts the Geometry System on a change of heading or change in speed so that TTA can be recalculated.

Event System - Holds the main event queue of the simulation, acting as a sorted priority queue where priority is time (though under the hood this is implemented as a dictionary time: set of events) to allow for quick removal and insertion of events. It forwards events generated at the current time directly to the main event loop, skipping the queue. It adds the SimStart and SimEvent to the queue during the verifying phase.

Main Event Loop - Loads the events for a given time and passes them to the relevant systems. It tracks which systems are actively working; when all systems have finished processing, it requests the next event from the event system, advancing the sim time to the time at that event. When it hits the SimEnd event, it waits for all systems to finish processing the current timestep and then shuts down, ending the simulation.

Legend - Standard System Init Proceedure

Throughout development a design pattern emerged that was common to most of the implemented systems:

Initializing

Save parent address (SimDriver)
Request address of necessary systems from SimDriver
Verify data
Wait for Events

GetEvent

Add event to ActiveEvents set
Process event
Alert Main Event Loop if additional systems will be activated to process new events i.e. a GoTo requires the Geometry System
Remove event from ActiveEvents
If there are no ActiveEvents, inform teh Main Event Loop that the system is done processing

Legend - Supporting Utilities

There are several supporting utilities which are used commonly through the Legend system.

Validator - A class containing methods for ETL proceedures and several data validation operations, usually invoked when constructing a datastruct object.

Random_Generator - A class that uses python's random functions and is designed to allow repeatability of sim results via seeding. Unfortunately, it was discovered during development that Python's set removal operations do not obey random seeding. This will require modifying the set class to override its element selection and removal code; once that change is in place, the Random_Generator will allow repeatable sim results.

Logger - A feature complete logger that logs at different warning levels to different file locations as set in the Sim.config file. It will also stream data to a server. Note that all of the output files are designed for ease of use with excel during post processing.

Run_time_utils - This class holds the sim start and end time as well as convenience methods for parsing time and file headers.

Haversine - This class was taken verbatim from the Python haversine package in order to reduce dependence on external modules. This allows us to assume a round earth in our distance calculation, averaging out the radius of the earth to 6371 km.

Legend - Next Steps

The next major feature for the Legend software is messaging, to allow entities to receive messages on events i.e. “arrival” and to throw message to each other.

After that, the next step is implementing entity finding via a divide and conquer strategy using locality workers that correspond to MGRS grids. The benefit of the MGRS grid system is two fold:

1. Simple geometric distance calculations (instead of great circle calculations) can be performed within a single grid square, allowing for a massive speed up in geometry calculations.

2. The predictable size of grids allows us to limit searches based on distance to only N number of localities where N is the maximum number of localities reachable by the search distance. This represents a huge improvement over system wide (global) searches.

Lastly, implementing resources in conjunction with messaging and finding would allow us to have entity behavior change based on available resource reserves and resources available in the environment.

Legend - Post Mortem

The Legend project presented an opportunity to touch on several hard problems including building a distributed simulation, maintaining state consistency across a distributed system of actors that do not share memory, logging the events of these systems in a sane manner that reflects both wall clock and sim time as necessary, deconflicting and sorting messages from distributed systems in a performant manner, and debugging a distributed system, while making the user interaction sane and intuitive.

The Entity Component System model implemented on top of the Actor model served to make this a surprisingly intuitive undertaking which I do not believe could easily (and certainly not sanely) have been implemented using a traditional threading model.

Initially,the system was supposed to be implemented in Scala to take advantage of the JVM performance, the rich ecosystem of Java libraries, and the ease of actor integration that comes from the Akka library. In practice, I found that my development rate became abysmal as I struggled to implement parity between disparate objects and actor system communication handling in an intuitive way. In the end, in order to have cleaner, more readable code and to increase agility of development, the decision was made to switch to Python after considerable development effort had already been spent.

I only regret that I did not make the decision sooner. Scala is a fantastic language and has considerable strengths, but the amount of extra syntax, classes, exception handling, and its relatively poor stack tracing meant that it was laborious to implement the distributed system and exponentially difficult to debug. Being proficient, but not having total mastery of the Scala language, I felt that if I had continued to work in the language the result would have been cumbersome and non-extensible.

Overall, I am pleased with the result, but time constraints have prevented me from implementing additional features. That said, the code base is fairly well organized and I found that implementing new features on top of the core system was not an overly arduous task. It is my estimate that I could have most of the additional features implemented and optimized with approximately one hundred more hours of work. Once these additional features are implemented the system will be comparable (and in some ways far superior) in capability to existing, commercially deployed ground truth simulation software i.e. ATLAS