2019 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
MODELING & SIMULATION, TESTING AND VALIDATION (MSTV) TECHNICAL SESSION
AUGUST 13-15, 2019 - NOVI, MICHIGAN

MoVE: A Mobility Virtual Environment for Testing Multi-Vehicle Scenarios

Marc Compere, PhD[1], Kevin Adkins, PhD[2], Otto Legon[1], Patrick Currier, PhD[1]

[1]Mechanical Engineering Department
[2]Aeronautical Science Department
Embry-Riddle Aeronautical University, Daytona Beach, FL

ABSTRACT

This paper presents a Mobility Virtual Environment (MoVE) for testing multi-vehicle autonomy scenarios with real and simulated vehicles and pedestrians. MoVE is a network-centric framework designed to represent N real and M virtual vehicles interacting and possibly communicating with each other in the same coordinate frame with a common timestamp. The goal is to provide a spectrum of test options from simulation-only to semi-virtual, to all real vehicles and pedestrians. A multi-vehicle test fidelity metric is defined that captures scenario realism more accurately than traditional hardware-in-the-loop style terminology.

MoVE's simple built-in vehicle models are described that provide positions in both latitude and longitude and Cartesian UTM XYZ coordinates. Live GPS inputs from real people or vehicles allow both virtual and real vehicles to interact through the virtual environment. Test results are presented from three experiments with real and virtual vehicles and pedestrians on a university campus. MoVE is open source software that is freely available in source code at Gitlab.com.

1. INTRODUCTION

Autonomous vehicle researchers around the world are pursuing multi-vehicle scenarios but lack a common framework in which to develop, test, and collaborate. Difficulty with testing even a single autonomous vehicle, aside from common technical challenges include vehicle registration, roadway, waterway or airspace restrictions, and the obvious safety risks of vehicle-to-vehicle or vehicle-to-pedestrian collision avoidance. The autonomy research community needs a way to test multiple autonomous vehicles along a realism spectrum from simulation-only to all real vehicles and real people. Multiple vehicles and pedestrians must be accommodated in simulation with clear steps for testing with real vehicles and real pedestrians in the real world.

Multi-domain testing with ground, surface and aerial vehicles also pose challenges not often faced in a single domain. Safety and legal limitations for experimental ground vehicles vary by state, not to mention the inherent risk with testing a passenger vehicle's pedestrian avoidance system. FAA regulations restrict unmanned aerial vehicle (UAV) flight to underneath 400 feet and place additional restrictions on operations within proximity of airports

and other controlled airspace. Boat registration for an unmanned surface vehicle can be challenging because the state Department of Motor Vehicles may not have procedures for registering unmanned vehicles. Maritime law lacks clarity for autonomous vehicles and, as with the automotive industry, correct behavior while navigating among other vehicles is sometimes unclear, as is liability during accidents. So there are legal and regulatory challenges for vehicles.

This paper presents a Mobility Virtual Environment (MoVE) for testing autonomous system algorithms, multi-vehicle scenarios, and their interactions with real and simulated vehicles and pedestrians. The result is a network-centric framework designed to represent N real and M virtual vehicles interacting and possibly communicating with each other in the same coordinate frame with a common timestamp.

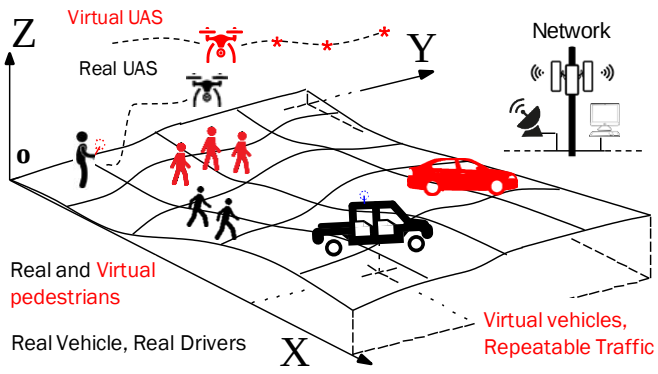

Figure 1: MoVE test scenario with real vehicles, real drivers, and real pedestrians combined with virtual vehicles and virtual pedestrians to improve autonomous technology development.

A literature review of comparable autonomous vehicle softwares is presented along with MoVE runtime concepts, interfaces, and test results from three test scenarios. The first scenario is entirely virtual to demonstrate the *avoid()* and *stay-in-bounds()* behaviors. The second test test is a traffic wave simulation with 4 real vehicles demonstrating traffic waves, or the accordion-effect. The third test is a medical evacuation scenario with all real vehicles and pedestrians including a ground vehicle, a pedestrian acting as an injured soldier, a medic, and a UAV flying a search pattern to report an injured soldier's location. Time-history and spatial plots are presented with vehicle-to-vehicle distance calculations indicating potential collisions. Plots on Google Maps using the lat/lon coordinates provides clear map-based positions and histories for all vehicles. Post processing or real time map plotting is available for analysis or runtime coordination using the Bokeh real time mapping interface. A configuration file allows researchers to experiment and share experimental setups like initial conditions, number and type of vehicles and custom behaviors easily. The MoVE set of concepts and interfaces are implemented as open-source software, written in Python and freely available at Gitlab.com.

1.1 Hardware-In-The-Loop is Inadequate

Model-based design methods have developed over the last several decades along with terminology intended to convey a mix of real and virtual components. Terms like hardware-in-the-loop (HIL, HWIL), model-in-the-loop (MIL), software-in-the-loop (SIL), soldier-in-the-loop, controller-in-the-loop (CIL), and even powertrain-in-the-loop (PIL) all assume a single, or primary device under test. In each of these terms, 'the loop' is intended to convey an interaction between one element that is virtual or simulated or emulated and the other that is real. Subtleties exist among different communities of practice. These terms are inadequate to describe the various elements in a multi-vehicle test with real and virtual vehicles and pedestrians.

To more accurately describe the various degrees of realism in a multi-vehicle scenario, a Testing Fidelity Metric (TFM) is defined in Table 1. A multi-vehicle scenario can be classified using values of 0-3 in five categories: (a) vehicle, (b) sensors, (c) algorithms, (d) environment, and (e) pedestrians. A scenario's Test Fidelity can be expressed as either a quintet of values (ex: 3/2/2/1/2) representing the categories in the order vehicle / sensors / algorithms / environment / pedestrians or as a single number (ex: 10) that represents the sum of the values. This metric provides a method for comparing the realism of differing scenarios.

The first scenario illustrated in Section 12 is classified as Test Fidelity Metric TFM=(1/1/1/1/0), or 4. This

represents a low realism metric for the all-simulated scenario. Next, test 2 in section 12 is the traffic wave Scenario with TFM=(3/0/0/3/0), or 6. This has higher test realism than the previous despite zero sensors and full human driving operation. Lastly, test 3 in Section 12 is the highest realism with TFM=(3/3/0/3/3), or 12. The vehicles are real, the sensors are real mobile phones reporting real positions, the scenario algorithms are a result of human operators, the environment is real, and the pedestrians are real. This is the highest realism metric of the three scenarios despite both vehicles being human piloted. The maximum possible Test Fidelity Metric score is 15, corresponding to all real vehicles, sensors, algorithms, and environment with pedestrians.

Table 1: Testing Fidelity Metric

Category	Rating			
	0	1	2	3
Vehicle	None	All simulated	Partially simulated vehicles and/or control	Real vehicles with control
Sensors	None	All simulated	Simulated sensors fused with real sensors	Real sensors
Algorithms	None or human operator	Run off-board, not in real-time, tele-op	Run partially onboard, partially off-board, or semi-automated	Run fully onboard in real-time, or fully automated operation
Environ-ment	None	Fully simulated	Real environment with simulated features	Real environment with features present
Pedestrians	None	All simulated	Some real, some simulated	Real pedestrians

1.2 MoVE Overview

The MoVE software is composed of vehicle models that execute in separate computer processes. Each has simple behavior threads with a priority-based scheduler similar to Rodney Brooks' subsumption architecture. These behaviors command simulated vehicle mobility and each vehicle reports position and health status updates to another process called Core. MoVE Core aggregates all vehicle messages to construct a complete scenario representation, `State`, with all vehicles and pedestrians, real or simulated.

Move Core computes distance calculations, notifies vehicles at risk of collision, provides vehicle to vehicle communication for *followMe()* or *searchAndRescue()* behaviors, and logs all vehicle data for after-action review. MoVE Core also outputs scenario `State` information to Bokeh, an open-source 2D visualization package, and MongoDB, an open-source database for test recording and playback.

1.3 Simple Vehicle Models and Coordinate Frames

The vehicle models use a READY-SET-GO-PAUSE-STOP state machine, an RK4 solver with simple mobility ODEs, a drift-free soft-real-time subsystem, and priority-based behavior scheduler to provide interesting mobility during simulation-only testing. Coordinate frames include a body-fixed *xyz* frame for each vehicle, an inertial *XYZ* frame and standard UTM coordinate transformations between inertial *XYZ* and latitude and longitude from live GPS sources. Each vehicle, whether live or virtual, reports position in both Cartesian *XYZ* UTM coordinates and latitude and longitude in decimal degrees. This allows for uniform post processing scripts for either virtual or real vehicles in both coordinate frames.

1.4 Real versus Virtual Vehicles

The built-in vehicle model equations of motion (EOM) provide simple mobility in the body-fixed frame, *xyz*, that are subsequently transformed into the inertial frame, *XYZ*. These EOMs are only used for virtual vehicle motion. To represent real vehicles or pedestrians, the same vehicle model is used with a flag to designate it as a live-GPS-follower. This type of vehicle listens on a separate network port for GPS position updates from a real vehicle or pedestrian in the real world. In this way, the mobility in the virtual world is determined by real GPS updates from the real person or vehicle in the real world. This approach allows virtual vehicles and real vehicles or pedestrians

to be represented in a common coordinate frame with a common timestamp. Virtual vehicles are provided awareness of other virtual vehicles and the live GPS followers through Core message updates. A sense and avoid algorithm is computed in Core with thresholds triggering warning messages to the relevant vehicles. Similarly, *followMe()* and *searchAndRescue()* behaviors in vehicle models get updates from Core as needed. Real vehicles or pedestrians can be made aware of other live or virtual entities in a similar way but this has not yet been implemented. Updates to live GPS followers is an active area of development using a web interface, a mobile device app, or an Augmented Reality (AR) headset.

1.5 Cellular Network Use for Testing

Live-GPS-followers stream GPS position from mobile phones or custom microcontrollers over the cellular network. The cellular network is the most convenient, widely available network for testing purposes. With the upcoming 5G network, this type of testing over the cellular networks will improve.

1.4 The Ideal Multi-Vehicle Development Pathway

A multi-vehicle simulation environment would, by itself, be useful for very low TRL experimentation, but it is important to have a direct pathway to incorporating real vehicles to improve test fidelity. Once the proof-of-concept is achieved in simulation, advancing the test maturity level by including real hardware in the same semi-virtual test is the ideal pathway to maturing a technology. What is needed is a simulation environment designed with clear networked interfaces to incorporate real hardware as the algorithms and tests mature while keeping the high-risk elements represented virtually until safety risks are addressed. Eventually, greater realism and expense must be undertaken such as one of the federally designated autonomous vehicle test sites [1] or MCity [2]. But, prior to advanced testing with all real vehicles, what the community needs is an open-source, widely available testing framework for low TRL, multi-vehicle autonomy testing. The MoVE framework provides such a framework for multiple real or virtual vehicles and real or virtual pedestrians with a freely available software implementation on GitLab.com [3].

2. LITERATURE REVIEW

Single-vehicle software simulations such as ADAMS CAR [4], SimCreator [5], TruckSim and CarSIM [6] are primarily designed for medium or high fidelity single-vehicle simulations. These have interfaces for interacting in multi-vehicle environments but were not designed specifically for multi-vehicle scenarios. This literature review focuses on multi-vehicle, multi-user, multi-pedestrian environments.

First generation multi-vehicle and multi-person live, virtual, and constructive environments originated in the early and mid 1990s with DARPA's SIMNET [7] and the US Army's CCTT [8]. The concept of integrating multiple vehicles among physically separate distributed, networked simulation environments was revolutionary [7]. OneSAF [9], ModSAF, WarSim, Janus, JCATS and others may be considered second generation multi-user, multi-vehicle simulation environments with real soldiers interacting within the environment during scenario rehearsal. These typically use DIS or HLA [10] for networked communications and use simple physics based models. OneSAF has remained viable for over 20 years from 1996 to the present and is the U.S. Army's fully-featured live, virtual, and constructive, multi-vehicle, multi-site military scenario training capability. OneSAF's Product Line Architecture Framework (PLAF) [11, 12] outlines a comprehensive suite of components for creating scenarios, pulling parameters from military databases, executing live and virtual training missions, managing friend and foe forces, maneuvers, terrain databases, and also performing after-action review (AAR). Complexity in OneSAF, WARSIM, Janus and others is high with a focus specifically on military use-cases. MoVE attempts to capture the real time, multi-vehicle, multi-operator networked aspects and after-action review via post-processing but is much simpler with a smaller code base. MoVE is not specifically targeted at military scenarios but can be configured for such scenarios. MoVE has a 2D live graphical output similar to OneSAF's top-down battlespace view but is designed as an engineering tool for simulating and testing a spectrum of real and virtual

vehicles with real and virtual pedestrians. MoVE's goal is to provide autonomous vehicle researchers with the ability to rehearse complex multi-vehicle scenarios, develop autonomy algorithms, and test messaging structures among vehicles and pedestrians along a spectrum of realism from fully virtual to fully real. Massively Multi-player Online games (MMO or MMOG) may be considered the third generation of multi-vehicle, multi-user software environments that resulted from widely available high speed network access, sophisticated graphics hardware, and a vibrant, creative open-source software community. For example, the *iRacing* online game has impressive vehicle models, tracks, multiple drivers interacting in real time over high speed internet connections. *iRacing* is proprietary but even open source gaming softwares are not designed for testing autonomous vehicles.

An important project historically focused on ground vehicle robotics is ROS: the Robot Operating System [26]. The Robot Operating System (ROS) is an open-source collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior. The companion program, ROS-Military (ROS-M) program aims to create a central registry of defense-related robotic components surrounded by a community of practice with common processes, systems, and standards. Both ROS and ROS-M are important contributors to the robotics community and focus on a single platform. MoVE is not intended for a single robot, but rather intended specifically for multi-vehicle scenarios. ROS 2.0 addresses multi-robot systems [28] but does not appear to address pedestrians explicitly as part of the scenario, although each robot may have pedestrian avoidance incorporated. ROS-M or ROS-2.0 is a likely candidate for future integration with MoVE.

ROS has an active user community, however, one of the largest active user communities for an open-source robot project is ArduPilot [29,30]. Perhaps the most active open-source user community for simulating and flying real autonomous vehicles, the ArduPilot suite includes a ground control station, mission planner, and other functions that work on multiple aerial, ground, and surface (water) hardware platforms. ArduPilot focuses on operating a single vehicle but multi-vehicle coordination is supported. ArduPilot is open source software and a likely candidate for future collaboration with the MoVE software environment.

One of the most sophisticated software environments specifically intended for multi-vehicle autonomous vehicles is Microsoft's AirSim [13]. Airsim is an open-source, software for simulating multiple autonomous vehicles in a photo-realistic 3D environment. It uses Unity's Unreal Engine [14] to simulate ground vehicles and popular aerial vehicles with impressive visual realism. It captures shadows, weather and time of day effects. It is impressive and a non-trivial code to interact with. The graphics portion of AirSim drives a large portion of the code base and complexity to compile and run on specialized hardware.

3D Graphics Belong Elsewhere

Most, if not all multi-vehicle gaming or simulation environments include a high-fidelity graphics component intended to improve realism for an immersive gaming or simulation experience. This single design feature drives computing software and hardware requirements towards high performance CPUs, GPUs, and low latency input devices and displays. The investment in hardware and software setup time can yield significant, impressive benefits. But the financial, time, and cognitive load associated with these graphics environments is non-trivial. The need to develop and test simple behaviors with real or virtual vehicles should not be impeded by the substantial barriers-to-entry from high fidelity 3D graphics. In short, high performance 3D graphics and associated CPUs, GPUs, and displays should not be a prerequisite to simulating and testing autonomous vehicles, behaviors, and algorithms.

MoVE Design Philosophy

In contrast, the MoVE framework is conspicuously lacking in 3D graphics capability. MoVE is designed with a Unix or Linux philosophy of multiple small, focused programs designed to perform clear, simple functions. Clearly defined interfaces allow MoVE to

effectively represent multi-vehicle scenarios composed of real and virtual vehicles communicating, each with various behaviors, without the overhead of 3D graphics output. A live 2D map plotting function is available but not necessary to rehearse complex multi-vehicle scenarios. This allows autonomy developers to focus on behaviors, inter-vehicle messaging, or improving model fidelity without any need for 3D graphics hardware or software.

Simple desktop computers, mobile devices with cellular network connections, and even edge computing devices like Raspberry-Pi class computers can run MoVE and participate in creative, networked, multi-vehicle simulation and testing. Also, with such low computational overhead, high vehicle count scenarios can be explored such as modeling swarming or teeming behaviors found in bird flocks or fish schools. Or, some or all vehicles in the National Air Space (NAS) can be simulated or monitored with virtual UAVs inserted to test flight guidelines or safety scenarios between manned and unmanned aircraft.

The ADS-B Framework

The aerospace community is quite familiar with the ADS-B family of aircraft messages. ADS-B is also somewhat well known in the Unmanned Autonomous Systems (UAS) community. However, other technical communities interested in autonomy may be less familiar, so a brief explanation is warranted. ADS-B is a messaging approach for commercial aircraft to both send and receive messages from ground stations, satellite, or other aircraft. ADS-B_Out contains position information outbound from each aircraft to make the surrounding airspace and control towers aware if it's location. ADS-B_In represents messages back toward the aircraft with weather updates or other vehicle locations. The MoVE software uses a messaging concept somewhat similar to ADS-B but is by no means compliant or even close to complete. The analogy is intended to convey the concept of individual vehicles sending outbound position updates to a central receiver (i.e. MoVE Core) and vehicle inbound messages being conveyed back to each vehicle with important information useful for that vehicle, such as a vehicle to follow or avoid [15].

3. MOVE ARCHITECTURE

The MoVE framework is composed of a process launcher, a test manager for issuing `runState` commands, and simple vehicle models that are run as separate computer processes. Move Core communicates with the vehicle processes and outputs full scenario snapshots to a MongoDB database for scenario recording and playback. A top-down 2D mapping script shows all vehicle locations using Bokeh, an open source 2D plotting library.

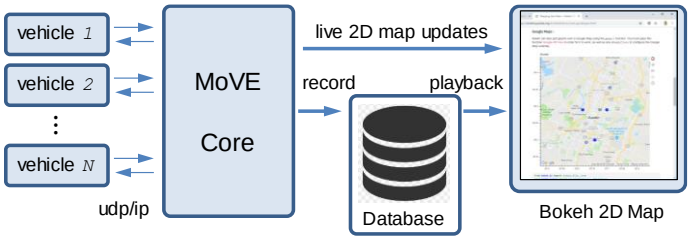

Figure 2: MoVE architecture with N vehicle models interacting with MoVE Core. Core sends State updates to a 2D map and database for test recording, post-processing and playback

4. MOVE RUNTIME DESCRIPTION

A test is initiated by launching Core, the vehicle models, and issuing `runState` commands to advance from `Ready`, to `Set`, then `Go`. A configuration file captures all relevant parameters like number of built-in vehicle models, number of live-GPS-follower models, numerical integration stepsize, udp/ip communication ports, and similar. Each vehicle model uses a drift-free soft-real-time subsystem to integrate mobility ODEs, each with it's own simulated time that advances at the same rate as wall-clock time. Each of these models is assigned a unique vehicle identifier, `vid`, so Core can distinguish different vehicles. All vehicles send updates via udp/ip to Core by sending position updates with velocity, orientation and vehicle health status messages. Core aggregates all incoming positions and health status messages into a single scenario representation called `State`. This `State` dictionary is polled periodically for detecting and avoiding collisions or providing position updates to other vehicles running the *followMe()* behavior. If two vehicles are at risk for collision, an avoidance message is sent from Core to both vehicles with position information to avoid collision. MoVE Core

records `State` for post-processing, playback, and After Action Review (AAR). MoVE Core has two simple command-line tools. The first launches vehicle processes and the second commands each vehicle to change `runState`.

Each vehicle process uses multiple threads for advancing the vehicle mobility dynamics in soft-real-time, sending and receiving udp messages with MoVE Core, and for behaviors controlling mobility or vehicle (model) functions. This architecture allows MoVE to represent a mixture of both real and simulated ground, surface, and aerial vehicles along with real or virtual pedestrians. For representing mobility of real people or real vehicles, some vehicle models are designated as live-GPS-followers that accept GPS latitude and longitude coordinates from a real GPS device. For live-GPS-followers there are no vehicle dynamics but, rather, the mobility is determined by the real GPS position updates in the real world, for example from a mobile phone or tablet. The widespread availability of low cost GPS receivers in smart-phone devices make capturing real vehicle or pedestrian movements straightforward. Open-source or low-cost apps are available for iPhone or Android devices to transmit GPS coordinates to the live-GPS-follower MoVE vehicles [16, 17]. If hand-held smart phone devices are too large or lack required accuracy, more specialized electronics can provide similar functionality with improved accuracy or smaller, lighter form factors for vehicles with limited payloads. A Cartesian XYZ frame is the underlying coordinate system in which vehicle model mobility is represented. All GPS latitude and longitude coordinates are converted to orthogonal XYZ Cartesian coordinates in meters via WGS84 UTM conversion [18].

Pedestrians are also important to capture in the scenario and may be either real or virtual pedestrians. MoVE mobility models can be configured as pedestrians with simple walking behaviors, ground vehicles with simple driving behaviors, or aerial or surface with flight or water navigation behaviors. A scenario is composed of multiple vehicles, pedestrians, and possibly multiple live-GPS-followers capturing movement of real people in the real world. Upon scenario execution, each vehicle model receives *Ready*, *Set*, *Go*, *Pause*, or *Stop* `runState` commands, communicates with Core, and possibly communicates with other vehicles. MoVE Core collects all communications and logs and sends periodic updates to the MongoDB database and Bokeh 2D visualization.

5. LIVE MAPPING

MoVE Core gathers all vehicle telemetry and periodically sends vehicle position updates to a Bokeh-based plotting program running on the same or a networked computer. This allows all virtual and real vehicles to be monitored in real-time on a 2D map with Google Maps satellite overlays. The map updates in real-time which displays new positions each time MoVE sends a new UDP message. The last N points of each vehicle are displayed to show tails of the recent position history. This allows the user to see a portion of the course of the vehicle leading up to the current position, and to identify any outliers in the data stream. A Google API Key and a GPS origin are required for live vehicle display. Example map with 16 virtual vehicles is shown in Figure 4.

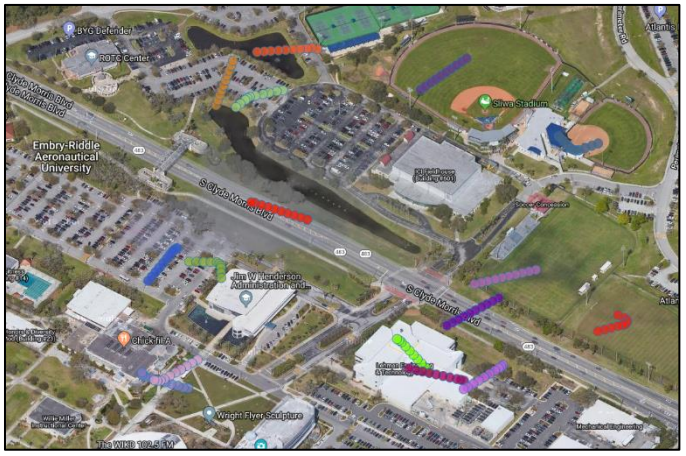

Figure 3: Map from the Bokeh-based live mapping program

6. VEHICLE BEHAVIORS

To provide interesting motion from simulated vehicles, a priority-based behavior scheduler selects different mobility behaviors similar to Rodney Brooks' subsumption architecture [19]. These

behaviors provide reactive, simple motions with low computational overhead and little or no world model necessary. The table in Figure 4 outlines prioritized behaviors that remain dormant until activated.

idx	behavior	priority	enable?	steer cmd	throttle	pitch cmd
0	wander	1	(0/1)	0	const	0
1	periodicTurn	2	(0/1)	rand[-1,+1]	-10%	0
2	periodicPitch	2	(0/1)	0	const	rand[-1,+1]
3	stayInBounds	4	(0/1)	turnAround	+10%	0
4	avoid	10	(0/1)	turnAway	const	turnAway
5	searchAndReport	0	(0/1)	N.A.	N.A.	N.A.

Figure 4: Table of behaviors and associated steer, throttle, and pitch commands chosen by the priority-based behavior scheduler

Behaviors in built-in models include *wander()*, *periodicTurn()*, and *periodicPitch()* plus the more complex behaviors named *avoid()*, *stayInBounds()*, and *searchAndReport()*. Each behavior is implemented as a thread within the vehicle model process and activates based on time or state conditions. The *stayInBounds()* behavior requires a bounding box, or convex polygon and associated point-in-polygon test to periodically check if the vehicle has exceeded the boundary. The *avoid()* behavior may be simple or complex based on vehicle, sensor, computing, or infrastructure capability. The nominal *avoid()* implementation relies on MoVE Core to provide vehicle-to-vehicle distance calculations and notify vehicles of problem conditions. The avoid behavior is one of the primary motivating reasons for developing a mixed virtual and real vehicle mobility testing capability. Sense-and-avoid is an important class of problems in unmanned aerial systems development [20]. One reason the sense-and-avoid behavior is difficult to test with real vehicles is the complexity of hardware, software and test conditions. The MoVE testing framework and freely available open-source software addresses this need.

The *searchAndReport()* behavior provides an interesting combination of simulation-only simplicity and real-world realism. This particular search function has no priority and no actuator commands. Its role is only to monitor the vehicle's current position and signal when it is within some threshold radius of a given target. The target's position is provided via MoVE Core and represents unrealistic information that a real-world search effort would not have. By definition, a real search effort has no knowledge of the unknown target location. But, a real vehicle could discover an unknown target with an on-board sensor like a camera or IR sensor. Upon discovering the unknown location, a real vehicle could, indeed, signal the discovery and last-seen time and location. The *searchAndReport()* behavior can mimic this discovery message in a simulated vehicle by disallowing the model's mobility functions from knowing the unknown location. In this way, the *searchAndReport()* behavior allows simulation-based rehearsal of search-based scenarios without a real vehicle discovering a real target. Or a simulated vehicle could be configured to search for a live-GPS-follower, which is a real person in the real world. In this way, complex search functions can be rehearsed in simulation only or with a mix of real and virtual vehicles or pedestrians.

7. BUILT-IN MOBILITY MODEL

The built-in mobility model equations of motion are a simple kinematic formulation with velocity and steer angle as inputs. The vehicle velocity in SAE body-fixed coordinates is given by

$$\vec{v}^{xy} = \begin{pmatrix} v_x \\ b \cdot \dot{\psi} \end{pmatrix} \quad (1)$$

where v_x is the specified forward velocity, $b = L_{char}/2$ is the distance from the vehicle CG to the rear axle (for ground vehicles) and $\dot{\psi}$ is the vehicle yaw rate about the body-fixed z-axis. Yaw rate is specified with operator steer commands with

$$\dot{\psi} = \left(\frac{v_x}{L_{char}}\right) \cdot \delta \quad (2)$$

where δ is the operator steer command and L_{char} is the vehicle's characteristic length. For ground vehicles, this represents wheelbase. For aerial or surface vehicles this represents total longitudinal vehicle length. The mobility model is simple wiwth a characteristic length, L_{char}, intended to capture an

approximate vehicle size for ground, air, surface, underwater vehicles or pedestrians. For example, integer multiples of L_{char} are used for thresholding in the *avoid()* behavior to allow for a small UAV to avoid collision with a much larger, manned aircraft like a 787 whose L_{char} is much larger. Transforming this body-fixed xy velocity into inertial, or terrain-fixed velocity with

$$\vec{v}^{XY} = T^T \cdot \vec{v}^{xy} \tag{3}$$

provides inertial frame XY velocities suitable for numerical integration with a fixed-step 4th order Runge-Kutta ODE solver. $T = T(\psi)$ and is the transformation matrix from body-fixed xy to inertial XY velocities give by

$$T = \begin{pmatrix} \cos(\psi) & \sin(\psi) \\ -\sin(\psi) & \cos(\psi) \end{pmatrix} \tag{4}$$

For aerial or underwater vehicles, elevation can be included by specifying vertical climb velocity with:

$$v_Z = v_x \cdot \sin(\theta) \tag{5}$$

where v_Z is the inertial frame vertical velocity as a function of θ, the operator's commanded pitch angle. These ODEs represent simple kinematic mobility models for a ground vehicle with Ackermann steering on flat ground. The simple dynamics includes a simple pitch equation to increase or decrease elevation, so these may also simply represent a fixed-wing aerial vehicle, rudder-steered surface vehicle, or even an underwater vehicle. Higher fidelity models can be incorporated using the same `runState` commands and the same MoVE Core incoming and outbound messages. Open-source examples are freely available at the MoVE software repository [3]. The GPL v3 license allows use for personal, research, or commercial use as long as the original source code is referenced and modifications are made available to the wider open-source community [21].

8. VEHICLE MODEL ARCHITECTURE

The vehicle model architecture is illustrated in Figure 5.

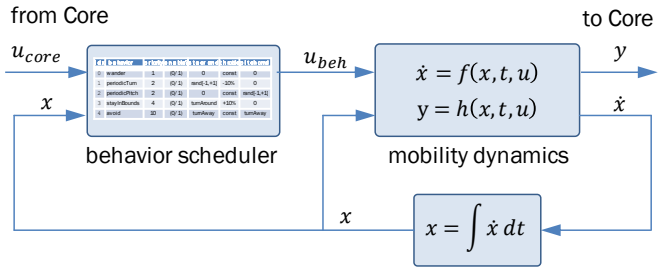

Figure 5: MoVE built-in mobility model with priority-based behavior scheduler

The layout uses a standard control diagram with operator inputs from MoVE core, a priority-based behavior scheduler to provide interesting operator commands, a drift-free soft-real-time subsystem, a 4th Order Runge-Kutta ODE solver, and simple mobility equations of motion. Vehicle model outputs include position in XYZ coordinates, positions converted to latitude and longitude, health status metrics such as soft-real-time margin, current behavior command, runState, unique vehicle ID, vehicle type and sub-type, and a vehicle name string for identifying vehicles during post processing.

9. SOFT-REAL-TIME SUBSYSTEM

A drift-free soft real time subsystem advances simulated time at the same rate as wall-clock time so simulated motion is realistic as it interacts with real vehicles or pedestrians. The MoVE software can be run on desktop PCs, laptops, Raspberry Pi class computers, or consumer grade tablets and mobile devices. Almost none of these computing platforms claims to provide a hard real time operating system, so the best and most convenient option is to achieve soft-real-time. Some simple implementations of soft-real-time delay a certain amount each loop resulting in simulated time advancing at roughly wall-clock-time. By subtracting the estimated computation time during each loop this can be reasonably effective. However, without an absolute reference this approach results in drift from wall-clock time, sometimes by many seconds or even minutes. The MoVE soft-real-time subsystem

captures the start time at execution and uses this absolute reference to ensure the N'th numerical integration step is taken and paused appropriately to maintain coordination with wall-clock time over many minutes and hours with a bounded total delay. The approach is similar to a PI controller with control error as absolute time difference and control input as sleep time.

The elapsed wall clock duration in seconds is $t_{wall\ clock} = t_{now} - t_{start}$. The start time, t_{start}, is captured when a vehicle process enters the GO `runState`. Total elapsed simulation time is $t_{sim} = N_{steps} \cdot h$ where N_{steps} is the total number of integration steps taken at the integration stepsize, h, in seconds. The difference between elapsed wall-clock time and total simulated time is the soft real time drift error:

$$dtErr = t_{wall\ clock} - t_{sim} \quad (6)$$

The amount the vehicle model process should delay, or sleep with a non-busy wait is the intended

$$sleep = \max(\ 0.0\ , cInt - dtErr - k_I \cdot dtErrSum) \quad (7)$$

The term $dtErrSum$ is the accumulated time errors much like an integral term accumulating error. The integral gain, $k_I = 0.3$ was chosen for stable delay time compensation limiting the total drift over many minutes with variable CPU load. Soft real time margin is defined as sleep time divided by communication interval. This allows quick reference on how much margin the soft real time process has for ensuring soft-real-time performance. As many vehicles (>100) begin to tax the computer's across all cores, the soft-real-time margin is a useful metric for understanding how well all vehicle models are advancing at wall-clock time.

10. SIMPLE VEHICLE BEHAVIOR MODELS

The mobility equations of motion accept operator inputs and solve for resulting motion. However, operator inputs must also be generated for N virtual vehicles. A priority-based behavior scheduler provide vehicle inputs within each vehicle model process to generate interesting motion in the virtual world. In the late 1980's and early 1990's Rodney Brooks and colleagues developed the subsumption architecture as a means of creating complex, fast, reactive robotic inputs to sensor stimulus [19]. At the time, the leading robotics philosophy demanded high world knowledge and extended computation to make thoughtful decisions [22]. The subsumption architecture is fast, has direct responses connected to certain inputs, is straightforward to program on embedded computers and yet can generate a rich result of complex behaviors, especially as the vehicle models interact with the environment and with one another, both real and virtual. At the heart of subsumption is a priority based scheduler which selects the winning behavior from the set of active behaviors. This competitive approach among behaviors is in contrast to Ronald Arkin's method which composes a blended approach from, perhaps, multiple active behaviors [23]. The MoVE built-in vehicle models are designed with a priority-based scheduler with in behaviors called *wander*, *periodicTurn()*, *periodicPitch()*, *stayInBounds()*, *avoid()*, and *detectAndReport()*. Behaviors under development include *followPath()* and *followMe()* for built-in vehicles.

11. TEST EXECUTION

A testing scenario typically has a common set of phases, each of which takes time to complete so MoVE's state machine transitions the `runState` variable suitable to each of these phases. The following list briefly describes characteristics of the five run states, *Ready*, *Set*, *Go*, *Pause*, *Stop*:

- *Ready*, or `runState==1` is the default starting run state for each newly launched vehicle process. The *Ready* state tells each vehicle model to provide MoVE Core with a low frequency (e.g. 1Hz) status update with vehicle ID, `runState`, and health status information.
- *Set*, or `runState==2` commands each vehicle to assign initial conditions to vehicle location, velocity, and optional payload. *Set* increases network traffic by commanding all vehicle models to report standard position, velocity, and health status updates to MoVE Core at the runtime communication interval, `cInt`. A reasonable

value for `cInt` is 0.1(s) which provides 10Hz MoVE Core updates. This allows test operators to verify the physical communications channels are well suited for the combination of communication intervals and number of vehicles in the scenario. In particular, wireless communication channels can be verified during the *Set* state.

- *Go*, or `runState==3` commands all vehicle models to begin numerical integration of the governing equations of motion in soft-real-time. This causes each vehicle model process to simulate vehicle or pedestrian motion in the virtual environment and continue to report position, velocity, and health updates to MoVE core at `cInt`. In the *Go* run state, live-GPS-follower models begin listening for their respective GPS locations on a pre-defined udp port. Upon receiving GPS location, this latitude and longitude is converted from lat/lon in decimal degrees to UTM coordinates which are meters in an orthogonal XYZ frame.
- *Pause*, or `runState==4` commands all vehicle models to stop numerical integration and hold position while test operators make adjustments to real hardware or evaluate the current test.
- *Stop*, or `runState==5` commands all vehicle processes, whether simulated or live-GPS-followers to close log files and exit gracefully.

12. SCENARIO EXAMPLES

Three scenarios are presented below that illustrate real and virtual vehicles interacting in a common coordinate frame with a common time stamp.

12.1 Scenario #1: All simulation, 3 vehicle

The first experiment presented is simulation-only with three virtual vehicles configured with *wander()*, *periodicTurn()*, *stayInBounds()*, and *avoid()* behaviors. The boundary imposed is a 100m x 100m region and each vehicle detects when it leaves the region and performs a turning maneuver, steering toward the center to return inside the boundary. Figure 6 shows the Google Maps view with latitude and longitude traces of all three vehicles, plus the boundary. The latitude and longitude origin is provided at the baseball field center.

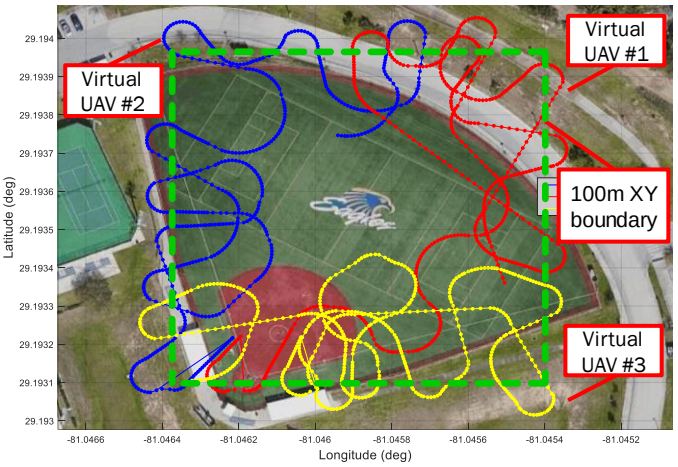

Figure 6: Three virtual vehicles in a 100mx100m region demonstrating *stayInBounds()* and *avoid()* behaviors

Figure 6 clearly shows the *stayInBounds()* behavior for each vehicle model while Figure 7 illustrates all vehicle distances to all other vehicles remained above a safe threshold.

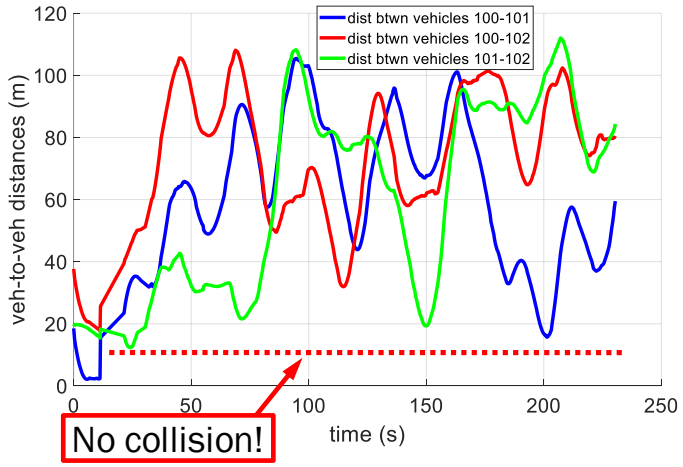

Figure 7: Three virtual vehicles in a 100mx100m region demonstrating *stayInBounds()* and *avoid()* behaviors. After startup (t=15s) MoVE Core directed all three vehicles to avoid the others as evidenced by the all-to-all distance plot versus time.

Aside from a close proximity startup between t=0 and t=15s, MoVE Core detected potential collisions and informed each pair of vehicles to take evasive action. The *avoid()* behavior is a simple example of sense-and-avoid. Virtual vehicle models integrate

ODEs in the body-fixed *xyz* frame, transform to the terrain-fixed *XYZ* frame with a UTM origin offset, and then use a UTM coordinate converter to output latitude and longitude for plotting on Google maps. The all-to-all distance calculations are performed in the inertial *XYZ* frame as a standard Euclidian 2-norm in units of meters.

12.2 Scenario #2: Traffic Wave Observation

The second experimental scenario is a traffic wave monitoring exercise with 1 (real) lead vehicle and 3 real followers. Four live-GPS-follower vehicles were configured along with 4 virtual vehicles that wandered and turned in the same scenario but did not interact with the live vehicles.

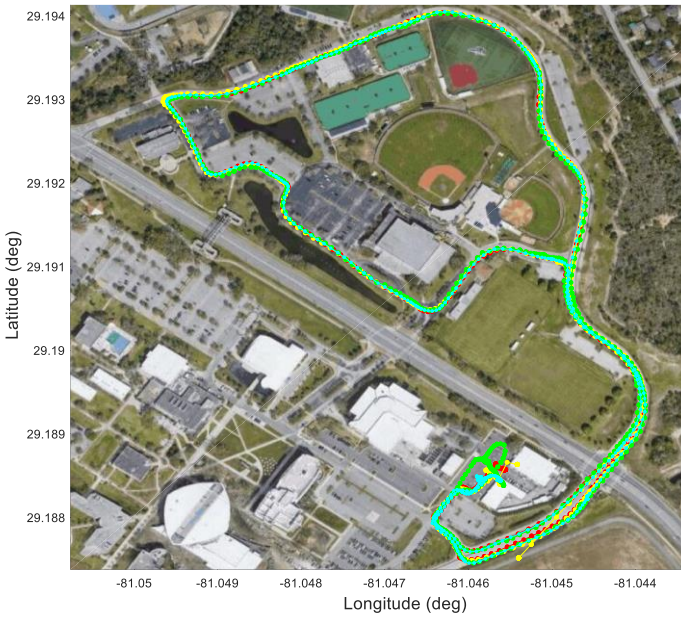

Figure 8: Traffic wave scenario with 1 lead vehicle and 3 follower vehicles streaming GPS position over the cellular network at 10Hz

The virtual vehicles are not shown but were present in the recorded data. All four Android devices running HyperIMU were configured to stream GPS location to MoVE Core at 10Hz over the cellular network [17]. HyperIMU is free for Android devices. Similar functionality streaming location position via udp over cellular networks will soon be available for iPhones using SensorLog [31]. Configuration and network communication was verified in runState 1 and 2 (*Ready*, *Set*). Then runState was assigned to 3 (i.e. *Go*) which initiated the testing sequence. The four vehicles drove from the campus parking lot to a city streetlight, then across the road to complete 1 lap around the university sports complex (Figure 8). The lap duration was just less than 6 minutes and 2.5km. The lap included approximately 7 speed bumps and stop signs causing the lead vehicle to intentionally accelerate and decelerate at each.

Post-processing the GPS locations of each vehicle in time consisted of UTM conversion from latitude and longitude in decimal degrees to orthogonal XYZ coordinates in units of meters. With distance converted to the inertial, or terrain frame, path lengths were calculated and plotted against time.

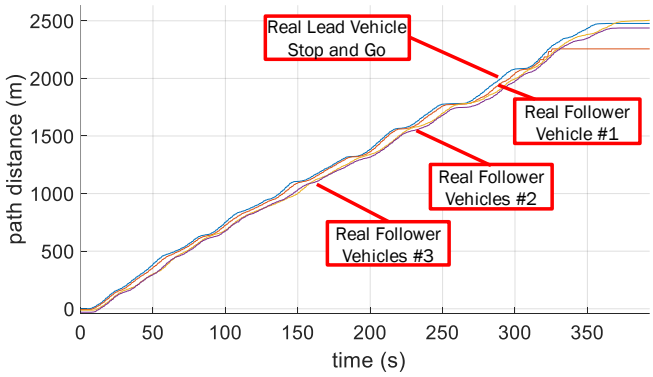

Figure 9: Traffic wave position versus time with 1 lead and 3 follower vehicles. Accordion or wave like motion is demonstrated.

MoVE successfully illustrates a common traffic wave pattern with real vehicles and real drivers. These time histories can be used to validate driver models, recreate traffic in simulation and ultimately design control laws for autonomous vehicles to reduce or eliminate traffic oscillation and traffic jams. Reduced traffic through autonomy is an important step toward reducing environmental impact, reducing fuel consumption, and reducing driver fatigue, distress and total drive time caused by highway traffic.

12.3 Scenario #3: Medical Evacuation

The US Army promotes a medical evacuation scenario as a useful situation for autonomous vehicles. The autonomous vehicle could enter a possibly hostile area, retrieve an injured person and provide automated medical evacuation from the danger area. Two real pedestrians and 2 real vehicles role played a medical evacuation sequence on a

university campus. The real person role playing an injured soldier went to a location and waited for the UAV to discover an injured soldier. then signaled help from the medic. The medic then approached the injured person, calling in an ambulance that traveled the boundary path to evacuate both medic and injured person back to the role-played hospital. The GPS traces collected by MoVE core from three live-GPS-follower vehicles is illustrated in Figure 10.

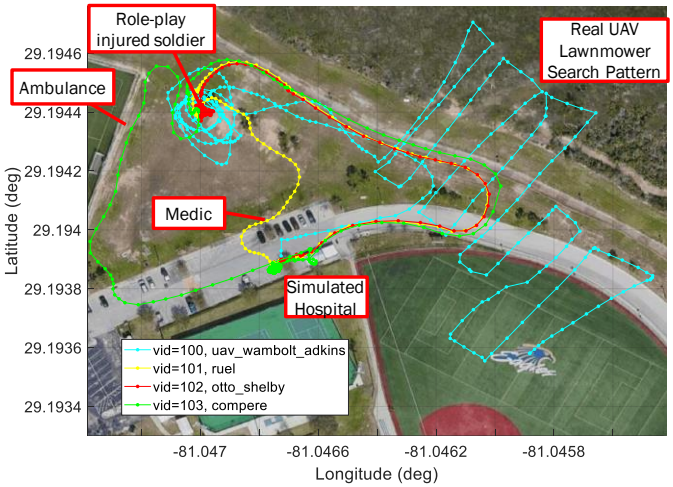

Figure 10: GPS Positions of 2 real pedestrians role-playing injured solder and medic, 1 real UAV flying lawnmower search pattern, and 1 real vehicle role-playing an ambulance.

All four traces are from live-GPS-followers. Two are from real people, one is from a real UAV flying, and the fourth is from a real vehicle traveling to retrieve both pedestrians.

Figure 11 shows the distance time history from all-to-all vehicles and pedestrians. The single trace with low distance near the beginning is the medic with Vehicle ID (vid) of 101 near the ambulance (vid=103). The first decreasing distance represents the UAV (vid=100) approaching the injured person (vid=102). The next distances that drop are the medic approaching the injured person and UAV. The third reduction in distances represents the ambulance approaching the medic, injured person with the UAV nearby, then taking the medic and injured person back to the hospital. The fourth rise and fall is the UAV following the ambulance returning to hospital. The UAV maintains safe distance from all others.

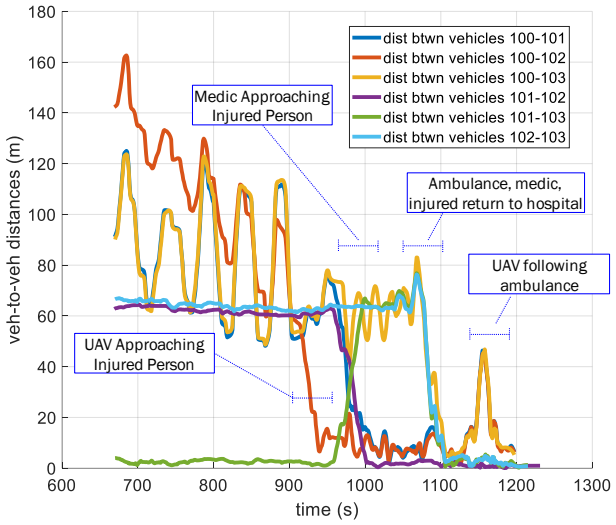

Figure 11: Distance traces for all-to-all vehicles with real UAV, 2 real pedestrians, and 1 real ground vehicle.

Post-processing multi-vehicle position histories can be a challenge, especially when the resulting logs have missing data points from temporary network drop-outs. One primary goal of the MoVE framework is to capture vehicle positions with a common timestamp. This is achieved but position updates are still individually collected at different, unique times. The concept of a vehicle-to-vehicle distance function presumes distances are able to be calculated at approximately the same time, plus or minus a few milliseconds. But because of intermittent packet loss over the cellular network, each position time history has a different array length which forces a time-based approach to computing vehicle-to-vehicle distances. The MoVE Core postProc script in Matlab negotiates these details and computes distances from each vehicle to each other vehicle for the logfile time ranges common to both. The number of vehicle-to-vehicle position calculations is $N_{distances} = (N^2 - N)/2$. The distance function executes fairly quickly on modern PCs when the number of vehicles, N, is small say, N < 300, and can be computed in real time or post-processed from logs.

The injured soldier, medic and ambulance are all role-played actors on a university campus. The role-played ambulance is a John Deere Gator XUV855D-S4 4-seater instrumented research vehicle and the UAV is

a DJI Matrice 100 quad rotor. This quad design has a large flat surface for mounting a sensor package.

Figure 12: LEFT: DJI Matrice 100 Quadrotor UAV with large surface for mounting instrumentation. RIGHT: Instrumented John Deere Gator research vehicle

UAV Camera images observing the medic and injured soldier is show in Figure 13.

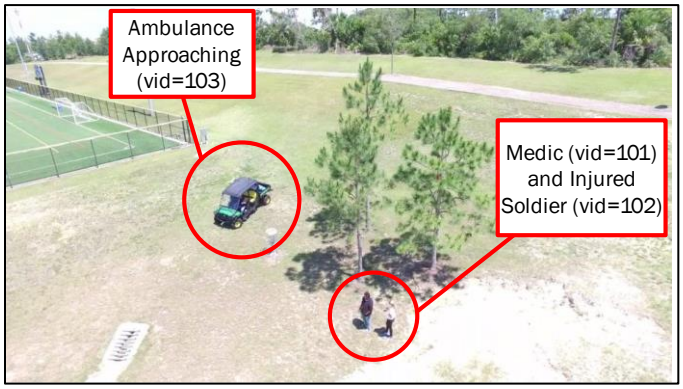

Figure 13: UAV Camera image observing ambulance approach medic and injured soldier on university campus

Figure 14 shows the UAV camera view following the ambulance back to the hospital.

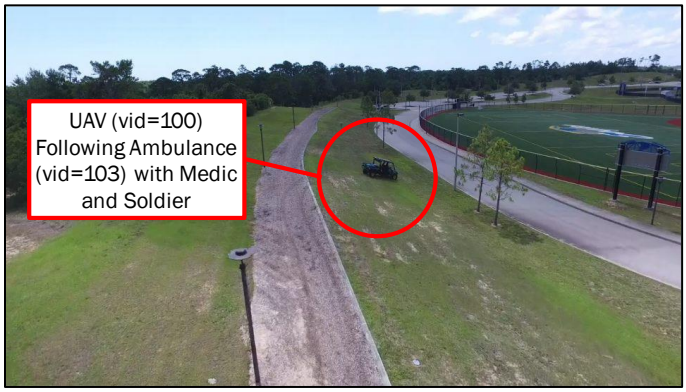

Figure 14: UAV camera image while following ambulance back to hospital on university campus

MoVE as Open-Source Software

The MoVE software development style follows the Unix philosophy of simplicity, transparency, modularity, and clearly defined interfaces [24]. Likewise, 3D graphics hardware and software is, and has always been a rapidly developing field with a substantial code base and literature base. Also, the complexity and effort required to develop and render visual databases is significant. For this reason, MoVE is entirely absent of 3D graphical output. MoVE provides real-time 3D position updates for each vehicle in the scenario, along with vehicle health status information via udp/ip netowork messages. The udp output messages are designed to interface with a 3D rendering engine, but there is no 3D real-time visualization tool included in the MoVE software distribution. Post-processing tools for visualizing 3D trajectories after a test are included. A real-time, 2D top-down position mapping tool based on Bokeh [25] is provided for monitoring MoVE vehicles with a Google maps overlay.

13. CONCLUSION

The need for multi-vehicle autonomy testing is clear and the autonomy community needs improved methods for testing, sharing, and comparing multi-vehicle scenarios. The Mobility Virtual Environment, or MoVE, provides one approach to data collection and sharing. A configuration file capturing all relevant settings can be shared among researchers for collaboration on scenario rehearsal, vehicle behaviors, or vehicle messaging. The MoVE software provides a mechanism for simulation-only testing and a clear mechanism for using a mix of real vehicles in the same scenario. The MoVE Core function provides test coordination with the *Ready*, *Set*, *Go*, *Pause*, *Stop* state machine and gathers all real and virtual vehicle positions for post processing afterward. Core also provides the sense-and-avoid detection and notification to vehicles. All-to-all distance calculations are performed by MoVE Core that detect imminent collision and notify both vehicles of interest. Bokeh is an open-source plotting library for 2D top-down display of real and virtual vehicles with a Google Maps overlay, in real time. A MongoDB interface records all vehicle messages for subsequence analysis and replay. The MoVE software

is written in Python 3 and is freely available on Gitlab.com for the autonomy and multi-vehicle simulation community to enhance collaboration.

14. ACKNOWLEDGEMENTS

The authors would like to thank Mr. Peter Wambolt for professional UAV flights and Mr. Ruel Morant and Ms. Shelby Bischoff for scenario rehearsal.

15. REFERENCES

[1] U.S. Department of Transportation. (2017, January 19). *U.S. Department of Transportation Designates 10 Automated Vehicle Proving Grounds to Encourage Testing of New Technologies.* Retrieved from Transportation.gov.

[2] University of Michigan. (2019). *Mcity Test Facility*. Retrieved from mcity.umich.edu.

[3] Compere, M. (n.d.). *Mobility Virtual Environment* publicly available open source software repository on gitlab.com: https://gitlab.com/comperem/move

[4] MSC Software. (2019). *Adams Car*. Retrieved from mscsoftware.com.

[5] Realtime Technologies. (2019). *Simulation for Autonomous Vehicle Testing*. Retrieved from faac.com.

[6] Mechanical Simulation. (2019). *Vehicle Simulation Products*. Retrieved from carsim.com.

[7] Cosby, L. N. (1995). *SIMNET: AN INSIDER'S PERSPECTIVE.* Alexandria: INSTITUTE FOR DEFENSE ANALYSES.

[8] Noble, J. L., & Johnson, D. R. (1991). *Close Combat Tatical Trainer (CCTT), Cost and Training Effectiveness Analysis (CTEA).* Fort Leavenworth: Department of the Army.

[9] Bowman, L., Rhinesmith, F., & Mervin, L. (2017). *Synthetic Environment Core (SE Core): 3D Geospatial M&S Summit Follow On.* Department of the Army.

[10] Hofer, R. C., & Loper, M. L. (1995). DIS Today. *Proceedings of the IEEE*.

[11] Smith, R. (2008). *OneSAF: Next Generation Wargame Model.* Department of the Army.

[12] Logsdon, J., & Wittman, D. (2008). *Standardizaton, Transformation, & OneSAF.* Orlando: Program Executive Office Simulation Training and Instrumentation.

[13] Kapoor, A., & Shah, S. (2017, November 13). *Microsoft extends AirSim to include autonomous car research.* Retrieved from Microsoft Research Blog.

[14] Wilson, J. (2019, April 2). *Unreal Engine 4.22 released.* Retrieved from Unreal Engine.

[15] Federal Aviation Administration. (2019, March 12). *Automatic Dependent Surveillance-Broadcast (ADS-B)*. Retrieved from faa.gov.

[16] Thomas, D. B. (2019, March 23). *SensorLog v2.6*. Retrieved from sensorlog.berndthomas.net.

[17] IANOVIR. (2019). *HyperIMU*. Retrieved from ianovir.com.

[18] Bieniek, T. (2017, April 6). *UTM*. Retrieved from pypi.org.

[19] Brooks, R. (1985). *A Robust Layered Control System for a Mobile Robot.* Arlington: Office of Naval Research.

[20] Angelov, P. (2012). *Sense and Avoid in UAS.* John Wiley & Sons.

[21] Smith, B. (2014, 11 8). *A Quick Guide to GPLv3*. Retrieved from GNU Operating System.

[22] Brooks, R. (1990). Elephants Don't Play Chess. *Robotics and Autonomous Systems*, 3-15.

[23] Arkin, R. C. (1998). *Behavior-based robotics.* MIT press.

[24] Raymond, E. S. (2003). The art of Unix programming. *Addison-Wesley Professional*.

[25] NUMFOCUS. (2018). *Welcome to Bokeh*. Retrieved from bokeh.pydata.org.

[26] ROS: The Robot Operating System, n.d., url: https://www.ros.org/about-ros/

[27] Jerry Towler, Matthew Bries, "ROS-Military: Progress and Promise", In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 7-9, 2018.

[28] ROS 2.0, "Why ROS 2.0?", url: https://design.ros2.org/articles/why_ros2.html

[29] Ardupilot Open-Source software, n.d., url: http://ardupilot.org/

[30] Open-Source Software Comparison Tool, n.d. url: www.openhub.net

[31] SensorLog iphone app, Bernd Thomas, url: apps.apple.com/us/app/sensorlog/id388014573