Proceedings of the SPIE Defense and Security Symposium, Orlando, FL, April 17-21, 2006.

MOAST and USARSim - A Combined Framework for the Development and Testing of Autonomous Systems

Chris Scrapper, Stephen Balakirsky, and Elena Messina

National Institute of Standards and Technology,
Gaithersburg, MD 20899, USA

ABSTRACT

Urban Search and Rescue Simulation (USARSim) is an open source package that provides a high-resolution, physics based simulation of robotic platforms. The package provides models of several common robotic platforms and sensors as well as sample worlds and a socket interface into a commonly used commercial-off-the-shelf (COTS) simulation package. Initially introduced to support the development of search and rescue robots, USARSim has proved to be a tool with a broader scope, from robot education to human robot interfaces, including cooperation, and more. During Robocup 2006, a new competition based on USARSim will be held in the context of the urban search and rescue competitions.

The Mobility Open Architecture Simulation and Tools (MOAST) is a framework that builds upon the 4-D Real-time Control Systems (4D/RCS) architecture to analyze the performance of autonomous vehicles and multi-agent systems. MOAST provides controlled environments that allow for the transparent transference of data between a matrix of real and virtual components. This framework is glued together through well-defined interfaces and communications protocols, and detailed specifications on individual subsystem input/output (IO). This allows developers to freely swap components and analyze the effect on the overall system by means of comparison to baseline systems with a limited set of functionality. When taken together, the combined USARSim/MOAST system may be used to provide a comprehensive development and testing environment for complex robotic systems.

This paper will provide an overview of each system and describe how the combined system may be used for stand-alone simulated development and test, or hardware-in-the-loop development and testing of autonomous mobile robot systems.

Keywords: Simulation, Control System, RCS, Intelligent Systems, Autonomous, Robots, Urban Search and Rescue

1. INTRODUCTION

Virtual (simulated) versions of NIST's Reference Test Arenas for Urban Search and Rescue Robots were developed to provide the research community with an efficient way to test algorithms independently of the costs associated with maintaining functional robots and traveling to one of the permanent arena sites for validation and practice. There are many aspects of development and research in which simulations can play a useful role. Simulation environments enable researchers to focus on the algorithm development without having to worry about the hardware aspects of the robots. If correctly implemented, simulation can be an effective first step in the development and deployment of new algorithms. Simulation provides extensive testing opportunities without risking of harm to personnel or equipment. Major components of the robotic architecture (for example, advanced sensors) can be simulated and enable the developers to focus on the algorithms or components in which they are interested. This can be useful when development teams are working in parallel or when experimenting with novel technological components that may not be fully implemented yet. Simulation can be used to provide access to environments that would normally not be available to the development team. Particular test scenarios can be

Further author information: (Send correspondence to S. Balakirsky)
S. Balakirsky : E-mail: stephen.balakirsky@nist.gov
C. Scrapper : E-Mail: christopher.scrapper@nist.gov
E. Messina : E-Mail: elena.messina@nist.gov

run repeatedly, with the assurance that conditions are identical each time. The environmental conditions, such as time of day, lighting, or weather conditions, as well as the position and behavior of other entities in the world can be fully controlled. In terms of performance evaluation, it can truly provide an "apples-to-apples" comparison of different software running on identical hardware platforms in identical environments. Simulations can also be used to multiply the number of robot platforms in order to study situations involving multiple vehicles. One example of this application of simulation environments is the human-robot interaction research of Lewis *et al.*[1]

Although simulation environments are important tools for the reasons just noted, they are not a panacea and must be used with an understanding of their limitations. Simulations cannot completely reflect the complexity and noise found in the real world. Simulated sensor feeds, even if degraded in order to avoid being unrealistically "clean," do not represent the type of input that a system will encounter from a real sensor in the real world. Similarly, communication subsystems have myriads of random degradations that cannot be accurately modeled in most simulations. Environmental models cannot capture the detail and variability present in the world. It is practically impossible to capture the level of detail of an environment (e.g., each blade of grass that a ground vehicle must go over) and the physical interactions that occur due to the modeling effort and the resulting strain on the processing power of the computer running the simulation. Similarly, the actuation of the robot cannot be completely modeled. Lags in response time, backlash in gears, and other physical symptoms, are difficult to characterize and accurately model. Therefore, developers must be aware of the limits of the simulation's power in validating and verifying their system's performance. It is an important and effective first step, but if a real system is to be built, much further testing will be needed in the real world.

2. BACKGROUND

Russell *et al.*[2] describe the the real world as a stochastic, non-deterministic, dynamic, multi-agent environment that is only partially observable by any one agent. Since virtual simulation environments can be made fully observable, they have long been used to assist in the development and testing of autonomous agents. They provide a safe, cost-effective alternative to real environments. However, in order to develop reliable, robust algorithms for autonomous systems, it is important to maintain the level of complexity and uncertainty found in real world environments.[3]

Dixon *et al.*[4] point out that there is a large amount of low-level infrastructure that is required to support the communication, control, and simulation of multiple agents in high-fidelity virtual environments. In addition, virtual simulation environments must provide tools and facilities that allow developers of autonomous systems to easily create worlds and to model robotic platforms and mechanisms that control the agent and entities in the simulation environment. Currently there are several simulation packages that assist in the development and testing of virtual simulation environments. Several of these packages are briefly discussed below.

2.1. Pyro

Pyro is a set of tools and facilities that adds a layer of abstraction between various simulation environments, robotic control languages, and researchers using them. By providing researchers with libraries of object code, Pyro acts as a single driver integrating robots, sensory suites, visualization tools, and simulator into a single Graphical User Interface (GUI). This standard Application Programming Interface (API) emancipates researchers from the nightmare of interfacing proprietary packages by providing a single user-friendly environment that controls a host of simulation tools.[5]

2.2. GameBots

GameBots provides high-fidelity simulation capabilities through an API that interfaces with the Karma game engine found in the first person shooter video game, Unreal Tournament. Essentially, it is a socket API to a controlled client/server network with distinct interfaces and protocols for inter-agent communication, virtual sensing, and operational command and control. Gamebots was initially developed to study human-computer interactions, artificial intelligence and multi-agent systems[6] .

2.3. Player

Player is a networked device interface that provides transparent command and control channels to multiple robots and their facilities. Player provides two complementary multiple robot simulators designed to work together interchangeably. Stage is a two dimensional simulator designed to handle a very large number of robots at low fidelity, where Gazebo is designed to be a high-fidelity simulator for smaller populations. Player is part of an open source community providing technical support and sample code for the simulation, testing, and evaluation of the mobile robots.[7, 8]

3. PROBLEM STATEMENT

An autonomous system can be viewed as an embodied agent, where the intelligent agent is housed within a mechanical or robotic system. The mechanical system is comprised of several subsystems that enable the agent to interact with the environment in different ways. In order for the agent to "act rationally," the agent may have to reason at a high level of abstraction to choose a behavior that is appropriate for a given situation such as "kick the ball". This high-level behavior must then be decomposed into a series of coordinated actions between some or all of the robotic subsystems.

Due to financial constraints, resource availability, and/or level of expertise, it is difficult to extensively develop, test, and analyze an entire autonomous system. This creates a situation in which developers develop and test their algorithms under a variety of conditions. This makes it difficult for developers to compare their own progress, and makes it extremely difficult to reproduce and compare the performance characteristics of different algorithms. Therefore it is important to develop a baseline simulation system, consisting of a baseline agent, common robotic platforms, and reproducible environmental conditions. This baseline system would limit the amount of variability in the development process, which is critical to gathering of performance characteristics on specific aspects of an autonomous system.

4. METHODOLOGY

The basic premise of this paper is to investigate a methodology for creating a baseline for measuring and evaluating autonomous systems with varying levels of cognition and different knowledge requirements, in realistic environments under repeatable trials with known ground truth, false detections, and noise.

While this may be done by treating the system as a simple monolithic module, it is the belief of the authors that the best means of tackling this problem is to decompose the system into a set generic modules with control interfaces that are able to fully control most aspects of robotic platforms. Implementing this baseline system as a hierarchical control structure distributes the computational complexities and compartmentalizes the responsibility and domain knowledge into a set of components operating at different levels of abstraction. Each echelon in the hierarchy can be further decomposed horizontally to create three categories of control: sensory control, mission control, and mobility control. In this decomposition, mobility refers to the control aspects of the vehicle that relate only to the vehicle's motion (e.g. drive wheel velocities), sensory control refers to systems that acquire information from the world (e.g. cameras), and mission controls are controllable items on the platform that are not related to mobility (e.g. camera pan/tilt or robotic arm).

Simulation environments have provided developers with a cost effective, safe environment to develop and test autonomous systems. Lately, there has been an emergence of game-based simulation environments that capitalize on the optimized rendering facilities and dynamic physics models contained in professionally developed game engines.[10]

Urban Search and Rescue Simulation (USARSim) was developed under a National Science Foundation grant to study Robot, Agent, Person Teams in Urban Search and Rescue.[1] This simulation package builds on a set of modifications to the Unreal game engine that enables actors in the game to be controlled through a TCP/IP socket API. USARSim's interface is designed to mimic the low-level robotic interfaces that are commonly found in USAR and Explosive Ordnance Disposal (EOD) robotics system. This provides developers with the embodiment needed for the development and testing of autonomous systems.

The MOAST framework provides a baseline intelligent system that is decomposed into encapsulated components that are designed based on the hierarchical 4-D/RCS Reference Model Architecture.[9] These components are vertically separable and horizontally decomposed into modular controllers of differing spatial and temporal responsibility for each of the three major subsystems. The MOAST framework is accompanied by a suite of knowledge repositories and tools that augments the intelligence of the baseline system and assists developers in the development, testing, and analysis of the each of its modules.

MOAST[11] and USARSim[12] are distinct open-source projects that can be found on SourceForge.net. With respects to the RCS hierarchy, USARSim provides the servo echelon, the robotic platform, and the environment. MOAST has implemented the RCS hierarchy from the Primitive Echelon to the Section Echelon. The combination of USARSim and MOAST provides developers with a versatile and comprehensive tool suite for the development, testing, and analysis of autonomous systems. It enables the tailoring of knowledge, the augmentation of the system though virtual sensing capabilities, the addition of extra external knowledge repositories, and a smooth transitional gradient between the purely virtual world and the real world implementation of autonomous systems. The remainder of this Methodology section will bore down into MOAST, then will continue to bore down into USARSim, and will finish with a discussion of an interface tool that seamlessly maps the two interfaces together.

4.1. Mobility Open Architecture Simulation and Tools (*MOAST*)

MOAST is a framework that provides a baseline infrastructure for the development, testing, and analysis of autonomous systems that is guided by three principles:

- Create a multi-agent simulation environment and tool set that enables developers to focus their efforts on their area of expertise without having to have the knowledge or resources to develop an entire control system.
- Create a baseline control system which can be used for the performance evaluation of the new algorithms and subsystems.
- Create a mechanism that provides a smooth gradient to migrate a system from a purely virtual world to an entirely real implementation.

MOAST has the 4D/RCS architecture at its core. The 4D/RCS hierarchy architecture that is designed so that as one moves up the hierarchy, the scope of responsibility and knowledge increases, and the resolution of this knowledge and responsibility decreases. Each echelon (or level) of the 4-D/RCS architecture is comprised of nodes which perform the same general type of functions: sensory processing (SP), world modeling (WM), value judgment (VJ), and behavior generation (BG). Sensory processing is responsible for populating the world model with relevant facts. These facts are based on both raw sensor data and the results of previous SP (in the form of partial results or predictions of future results). The world model must store this information, information about the system itself, and general world knowledge and rules. Furthermore, it must provide a means of interpreting and accessing this data. Behavior generation computes possible courses of action to take based upon the knowledge in the WM, the system goals, and the results of plan simulations. Value judgment aids in the BG process by providing a cost/benefit ratio for possible actions and world states.

The principal difference between components (referred to as nodes) residing at the same echelon is in the set of resources managed. The principal difference between echelons is in the knowledge requirements and the fidelity of the knowledge and planning space. This regularity in the architectural structure enables scaling to any arbitrary size or level of complexity. Each echelon within 4-D/RCS has a characteristic range and resolution in space and time. Each echelon has characteristic tasks and plans, knowledge requirements, values, and rules for decision-making. Every node in each echelon has a limited span of control, a limited number of tasks to perform, a limited number of resources to manage, a limited number of skills to master, a limited planning horizon, and a limited amount of detail with which to cope.

Module-to-module communications in MOAST is accomplished through the neutral messaging language (NML).[13] NML is currently ported to platforms that include Linux, Sun Solaris, SGI Irix, VxWorks, LynxOS,

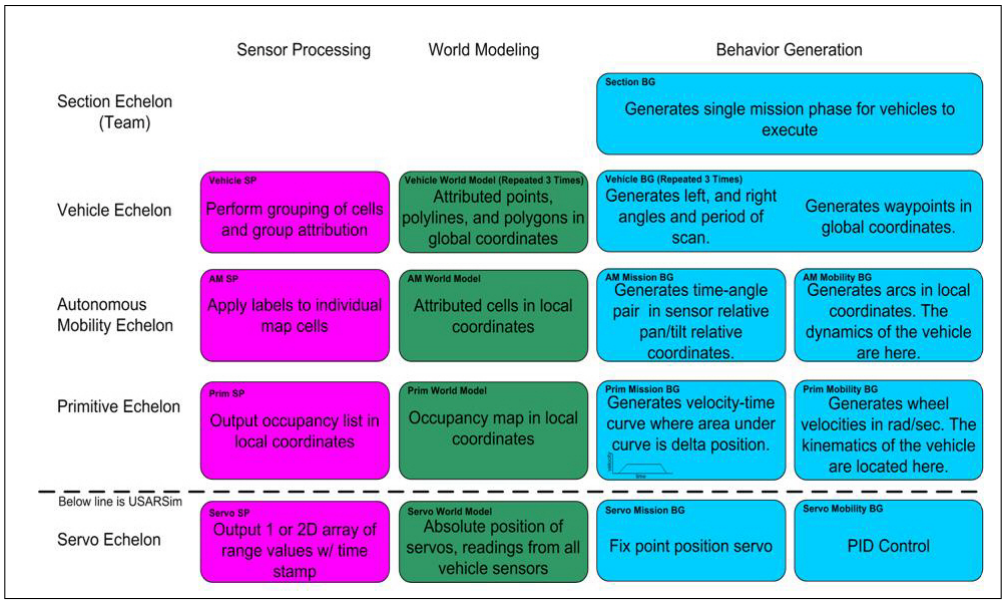


Figure 1: Modular Decomposition of MOAST framework that provides modularity in broad task scope and time.

QNX, Microsoft Windows, and Max OS X. NML allows applications running on one platform to communicate with ones running on any other platform. It is based on a message buffer model, with a fixed maximum size and variable message length. Buffers may be blocking or non-blocking, queued or non-queued, and may be polled or use a publish/subscribe model. The use of NML allows for a uniform application programming interface that is protocol independent. The system automatically delivers messages over one of the supported protocols that include shared memory, backplane global memory, TCP/IP sockets, and UDP datagrams while accounting for compiler structure padding, mutual exclusion, and format conversions. The specific communications protocols for each buffer are defined in a configuration file, not in source code.

The hierarchical decomposition of MOAST is depicted in Figure 1. Under this decomposition, the connection into the real or simulated system's API may be seen as fulfilling the role of the servo echelon, where both the mobility and mission control components fall under BG. The sensors are able to output arrays of values, world model information about the vehicle self is delivered, and mission package and mobility control are possible. This interface is described in detail in section 4.3. Section 4.1.5 will provide a synopsis some of the developmental tools coupled with MOAST. Section 4.1.6 will discuss the knowledge repositories used to augment the intelligence of MOAST. Section 4.1.1 through section 4.1.4 will concentrate on the functionality and interfaces of the each echelon moving down the hierarchy.

4.1.1. Section (Team) Echelon and Above

The highest echelon that has currently been implemented under the MOAST framework is the Section or Team Echelon, where only the BG node exists. This has the responsibility of taking high-level tasks and decomposing them into tasks for multiple vehicles. For example, the Section Echelon mobility may plan cooperative routes for two vehicles to take in order to explore a building. This level must take into account individual vehicle competencies in order to create effective team arrangements. An example of this would be commanding Section 1 to explore the first floor of a building and Section 2 to explore the second floor. Based on the individual team's performance, responsibilities may have to be adjusted or reassigned.

4.1.2. Vehicle Echelon

The Vehicle Echelon behavior generation is in charge of accepting a mission for an individual vehicle to accomplish and decomposing this mission into commands for the vehicle subsystems. Coordinated way points in global coordinates are then created for the vehicle systems to follow. This level must balance possibly conflicting

objectives in order to determine these way-points. For example, the Section Echelon mobility BG may command the vehicle to arrive safely at a particular location by a certain time while searching for victims of an earthquake. The Vehicle Echelon mobility BG must plan a path that maximizes the chances of meeting the time schedule while minimizing the chance of an accident, and the Vehicle Echelon mission BG must plan a camera pan/tilt schedule that maximizes obstacle detection and victim detection. Both of these planning missions may present conflicting objectives.

SP at this level connects to a geographic information services (GIS) database to extract simulated grouped features. These features include attributed points, lines, and polygons.

4.1.3. Autonomous Mobility Echelon

The Autonomous Mobility Echelon provides the ability to follow paths and compute maps of processed sensor data. Sensor processing exists to convert the instantaneous set of location-elevation triplets into a multi-attribute vehicle centric map in vehicle global coordinates. This map is of fixed size and is centered on the current vehicle location. As the vehicle moves, distant objects fall off of the map. Attributes of this map include elevation, range from the sensor to the location at last detection, number of detections over last set of cycles, and the probability that a location contains an obstacle. Obstacle determination is based upon a filtered version of the Primitive Echelon data.

The behavior generation is in charge of translating commanded way-points for vehicle systems into dynamically feasible trajectories. For example, the Vehicle Echelon mission controller may command a pan/tilt platform to scan between two absolute coordinate angles (e.g. due north and due east) with a given period. BG must take into account the vehicle motion and feasible pan/tilt acceleration/deceleration curves in order to generate velocity profiles for the unit to meet the commanded objectives.

4.1.4. Primitive Echelon

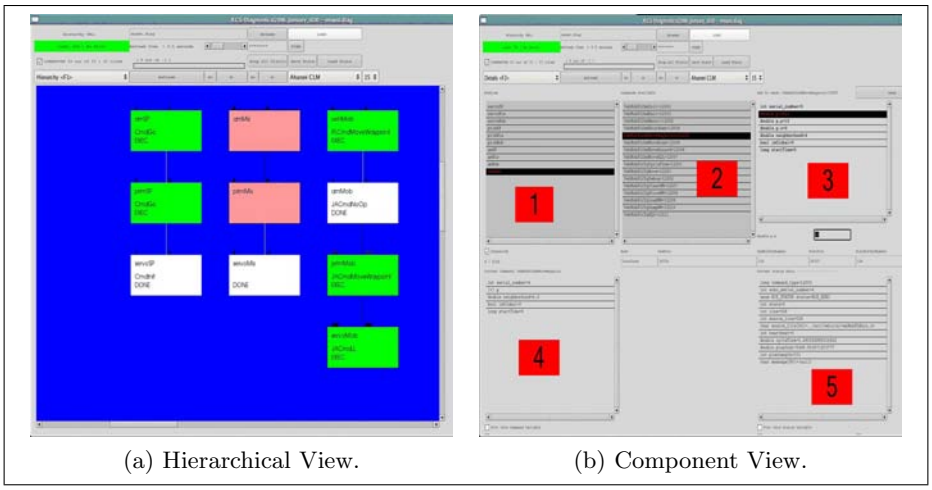
The Primitive Echelon interfaces with the autonomous agent's raw input and output signals. Sensor processing exists to read in the range/angle output of a laser range sensor and convert this into a set of location-elevation triplets with location given in vehicle global coordinates. The primitive echelon behavior generation expects a set of constant curvature arcs or constrained way points for the vehicle subsystems as input, and converts these into velocity profiles for individual component actuators based on vehicle kinematics. For example, the AM Mobility BG will send a dynamically correct constant curvature arc for the vehicle to traverse. This trajectory will contain both position and velocity information for the vehicle as a whole. For a skid steered vehicle, the Primitive Echelon BG plans individual wheel velocities based on the vehicle's kinematics that will cause to vehicle to follow the commanded trajectory. During the trajectory execution, BG will read vehicle state information from the Servo Echelon WM to assure that the trajectory is being maintained and will take corrective action if it is not. Failure to maintain the trajectory within the commanded tolerance will cause BG to send an error status to the AM Mobility BG.

4.1.5. MOAST Tools

The MOAST framework is packaged with several tools to assist in the development, testing, and debugging of MOAST compliant architectural components. These tools provide the ability to auto-generate code, to monitor command and control flow at the unit level and the hierarchical level, and provide implementation templates to assist the production of the computational components for the hierarchy.

Two different techniques are generally provided to exercise and test a control module. The first is a simple command line shell that allows specific commands to be sent to the module and status to be read. The second is a comprehensive Java-based diagnostics tool known as the RCS Diagnostic Tool that allows for complete module testing.

The Diagnostic Tool is a visualization tool that is able to monitor message traffic in the all the NML buffers in the hierarchy. This enables developers to get a snapshot of the states, status, and message traffic in the entire hierarchy. The Diagnostic Tool provides different views that enable a developer to monitor the entire hierarchy or a single node. In Figure 2a, the hierarchical view of the Diagnostic Tool is shown. This view provides the name of the node, the current command and the current state, and depicts the interfaces connecting each of the



(a) Hierarchical View. (b) Component View.

Figure 2: RCS Diagnostic Tool.

nodes in the hierarchy. The Diagnostic Tool provides a component view for each node as shown in Figure 2b. This view enables the user to view the details of the current command that any node is executing and allows the user to send any command to any node.

The Code Generation tool is a Java tool that utilizes the interface definitions, specified in a header file, and auto-generates the appropriate C++ NML buffer source code for that specification. This allows developers to concentrate their efforts on the development of well-defined interfaces without the burden of writing the underlying communication protocol. The RCS libraries also provide a robust collection of the coordinate representations and transformations that enable users to easy switch between different coordinate systems. In additional to the code generation tool, the RCS libraries provides several tools that assist developers in parsing initialization files and to perform various I/O operations.

There are other tools that have been compiled into MOAST, such as skeleton applications, administrative finite state machine diagrams, and sensor visualization tools that provide to assist in the development of autonomous systems. For more information of the tools provided by MOAST please refer to.[11]

4.1.6. Knowledge Repositories

Intelligence may be defined as "the ability to comprehend; to understand and profit from experience",[14] which implies the degree of intelligence is the ability to assert knowledge in the decision making process.[15] Therefore, in order for an autonomous system to demonstrate intelligence, the system must be able to apply past experience to its current situation. The MOAST framework allows this interaction by providing a set of flexible centralized knowledge structures containing *a priori* and *in situ* knowledge that is generalized, labeled, and grouped into relational networks. This knowledge is encapsulated in a central knowledge repository that is globally accessible and can be tailored to meet the knowledge requirements for specific modules through the development of inference engines and knowledge filters.

4.2. Urban Search and Rescue Simulation(*USARSim*)

USARSim is a high-fidelity physics-based simulation system that provides the embodiment and environment for the development and testing of the intelligence of autonomous systems. This is an open source simulation environment that is based on Epic Games Unreal Tournament 2004. Originally developed to study Human Robotic Interactions in multi-agent environment in an Urban Search and Rescue Environment,[1] USARSim is expanding its capabilities to provide realistic simulation environments to assist in the development and testing of cognitive systems, autonomous nautical vessels, and autonomous road driving vehicles.

USARSim utilizes the Karma Physics engine and high-quality 3D rendering facilities of the Unreal game engine to create a realistic simulation environment that provides the embodiment of a robotic system. The authoring

tool and programming language that are contained in Unreal enable developers to build and customize worlds and robotic systems that may be used within USARSim. (Currently, several models for robotic platforms and worlds are contained the USARSim distribution.) The embedded client-server architecture of the Unreal game engine enables USARSim to provide individualized control over multiple robotic systems through discrete socket interfaces. There interfaces provide a generalized representation language that enables the user to query and control the robot's subsystems.

4.2.1. USARSim Robot Interfaces

With respect to the embedded intelligent agent, the robotic platform is defined by its interface rather than the underlying mechanism that facilitate its actions. Therefore, it is important to consider the complexity of the interface, in terms of its organization, standardization, and extendibility, when designing its structure and representation. In order to create an interface that is both easily usable and adaptable, it is important that the representation of knowledge, whether input or output, be general, simple, orthogonal, readable, and complete. For instance, decoupling the interface from specific robots eases the learning curve by decreasing the semantics of the interface when adding new robots or subsystems. An interface that is able to work with a certain class of robot (e.g. skid steered) should be able to work with any robot of that class. However, different robots may have specialized attributes or functions that were not anticipated in the initial design. Rather than developing custom calls for these functions, it would be best to expand on the existing interface.

Therefore during the development of the interface to USARSim many factors were taken into account to ensure that the interface was both well-defined and standardized. This is done by decoupling the interface from the underlying simulation representation, and applying standards and conventions that are used within the scientific community. USARSim decouples the units of measurement used inside Unreal by ensuring that all units meet the International System of Units (SI Units),[16] a modern metric system of measurement. The set of base units that are defined are also the basis for the derivation of other units of measurement, e.g. m/s.

The coordinate systems for various components must be consistent, standardized, and anchored in the global coordinate system. USARSim leverages the previous efforts of the Society of Automotive Engineers, who published a set of standards for vehicle dynamics called SAE J670: Vehicle Dynamic Terminology.[17] This set of standards is widely used in the United States for vehicle dynamics, and contains illustrated pictures of coordinate systems, definitions, and formal mathematical representations of the dynamics.

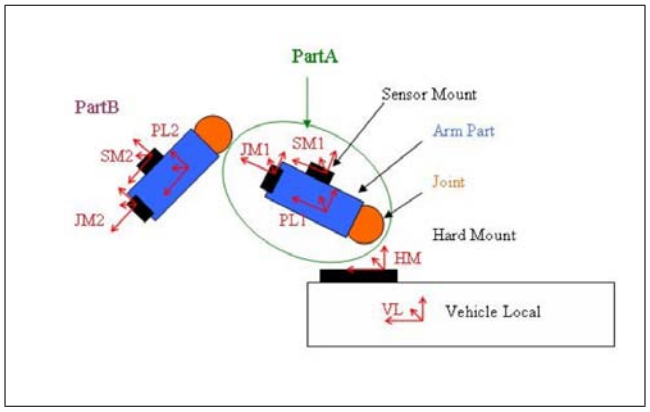


Figure 3: Internal Representation of an Robotic Arm.

Finally, the messaging protocol, including the primitives, syntax, and the semantics must be defined for the interface. Messaging protocols are used in USARSim to insure that infrequent and vital messages are received. The primitives, syntax, and semantics define the means in which a system may effectively communicate with USARSim, namely to speak USARSim's language.

There are three basic components that currently exist in USARSim: robots, sensors, and mission package. For each class of objects there are defined class-conditional messages that enable a user to query the component's

geography and configuration, send commands to it, and receive status back. This enables the embodied agent controlling the virtual robot to be self-aware and maintain a closed-loop controller on actuators and sensors.

Description of Class Conditional Messages for Robotic Arm	
Class Name	Description of Message Content
Configuration	Represents the components and their assembly with respect to each other.
Geography	Represent the pose of the sensor mounts and joints mount with respect to the part, and the pose of the part with respect its parent part.
Commands	Represents the movements of each of the joints, either in terms of position and orientation or velocity vectors.
Status	Represent the current state of the robotic arm.

Table 1: Description of class conditional messages for the internal representation of a robotic arm shown in Figure 3.

The formulation of these messages is based on an underlying representation of the object and includes their coordinate system, composition of parts, and capabilities. This highlights a critical aspect underlying the entire interface: the representation of the components and how to control those components. For example, examine a robotic arm, whose internal representation is visualized in Figure 3. In order for there to be a complete and closed representation of this robotic arm, the class conditional messages that are sent over USARSim must capture this representation. In Table 1, descriptions of the class conditional messages are shown to illustrate how this representation is captured.

4.3. Simulation Interface Middleware (*SIM-ware*)

One of the goals of MOAST is to provide a modular environment that provides a gradient of configuration from the purely virtual world to the real world. SIM-ware enables MOAST to seamlessly plug into any interface or API for a real or virtual vehicle. However, not all interfaces use the same messaging protocol, semantics, or exist at the same level of abstraction.

SIM-ware is an application that acts like an adapter, designed originally to enable MOAST to connect with real vehicle platforms and different simulation platforms. SIM-ware is made up of three basic components: a core, external knowledge repositories, and skins.

The core of SIM-ware is essentially a set of state tables and interfaces that enables SIM-ware to administer the transparent transference of the data between two different interfaces. Interfaces are being added to SIM-ware to augment the internal world model to provide virtual sensing and ontology access. The skins are an interface-specific parsing utility that enables the core translates incoming and outgoing messaging traffic to meet the appropriate level of abstract and data acquisition.

5. CASE STUDY

The Primitive Mobility BG module will be used in this case study to demonstrate how a developer can use MOAST and USARSim to develop, test, and evaluate algorithms for an autonomous system. In Section 4.1.4, a brief overview of the range and responsibility of the Primitive Echelon was provided. This section will provide an overview of the Primitive Mobility BG module's responsibilities, a description of the methodology used to develop a particular implementation of the Primitive Mobility BG module, and an illustration of the methodology used to quantitatively measure the performance of the algorithm in a repeatable manner.

5.1. Primitive Mobility BG Problem Description

The Primitive Mobility BG module is responsible for controlling the instantaneous velocity of a robotic system. Recall the MOAST architectural diagram in Figure 1. Notice that the Primitive Mobility BG module is located under the Autonomous Mobility BG module and above the Servo Mobility BG module. Therefore, the Primitive Mobility BG module must conform to the interfaces defined by MOAST that enable it to receive commands from

and send status to the Autonomous Mobility BG module, as well as send commands to and receive status from the Servo Mobility BG module.

The dynamics of the robot system are modeled in this module using *a priori* kinematic models and platform specific parameters obtained from the Servo Echelon Mobility BG module. The Primitive Mobility BG module receives a series of constant curvature arcs or path segments that are piece-wise continuous, and a set of arc tolerances for each path segment. Primitive Mobility combines this knowledge to plan the speed and the heading for the robotic platform. This is transformed further using dynamic models into actuator commands, which for the case of a skid-steered vehicle would be wheel velocities. The actuator commands are then routed to the appropriate Servo Echelon BG module.

In general, this is an over-constrained problem. This means that there are more constraint equations than there are parameters for those equations, and results in the condition that for a given assignment of values, a solution may not exist. Therefore, one or more of the assigned values must be treated as soft constraints that can be relaxed, or adjusted, in order to find a solution. Consider an example where both speed and heading are fixed constraints. If the current path segment is defined by a small radius, there might exist a speed that will cause the vehicle to tip over. To avoid this, either the speed must be reduced or the heading changed.

5.2. Primitive Mobility BG Implementation Methodology

In this implementation of the Primitive Mobility BG module the speed constraint is relaxed first in favor of the geometric constraints. This implies that a tight turning maneuver is not made wider to maintain the current velocity. Rather the motion of the vehicle is slowed down to maintain the appropriate heading.

After a drive command is received from the Autonomous Mobility BG module, the planning algorithm converts each of the constant curvature arcs and arc tolerances into a series of way points that are defined by a specific neighborhood. The planner then systematically computes the speed and angular velocity from one way point to the next using the equations formally expressed by:

$$\nu = \nu_{max} \left(1 - \frac{|\theta|}{\theta_{cutoff}} \right) \quad (1)$$

$$\omega = -1 * \omega_{max} \left(\frac{\theta}{\theta_{cutoff}} \right) \quad (2)$$

where ν is the speed, ω is the angular velocity, θ is heading deviation According to this set of equations, the speed, ν, is reduced and the angular velocity, ω, is increased as the heading deviation, θ, increases.

5.3. Primitive Mobility BG Measuring the Performance

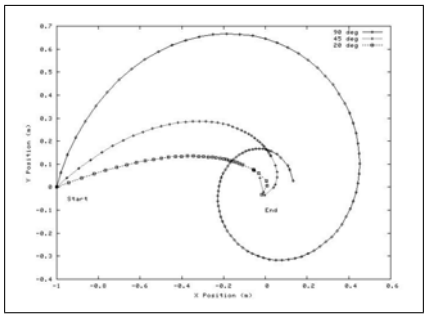
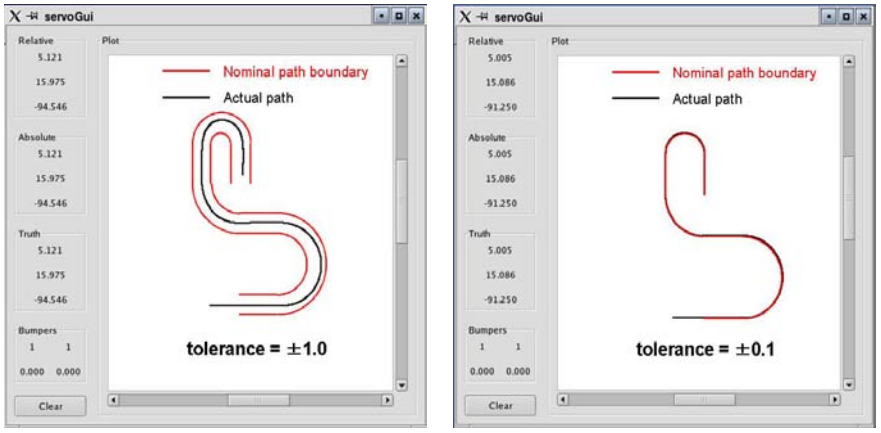


Figure 4: Plotted comparison of adjusting the cutoff angles

When analyzing this algorithm, it is wise to first examine how each of the parameters may affect the mobility and performance characteristics of the algorithm. For example in Figure 4, the effects of different cutoff angles are plotted on a graph for a given start and end point. Notice that algorithm tends to move toward the goal more tightly for smaller cutoff angles and depending on the systems mobility characterisitics, it may not be able

to achieve the goal if the cutoff angles is 90 degrees. However, smaller cutoff angles increases the slope of the velocity profiles and can cause abrupt changes in speed and heading.

This algorithm is made to honor the appropriate NML interfaces in MOAST, and is inserted into the MOAST architecture, as a new Primitive Mobility BG node. Using the RCS Diagnostic Tool, a developer can monitor the command follow through the hierarchy, making sure that no errors are produced by the insertion of the new module. The developer can then use the component view of the RCS Diagnostic Tool to conduct unit tests on the new module, verifying that the output is appropriate for a given input and tune the performance.



(a) Tolerance of 1.0. (b) Tolerance of 0.1

Figure 5: ServoGUI visualization of actual path vs. nominal path region

Now that the algorithm is tuned and is interacting appropriately in the MOAST architecture, it is time to quantifiably test the new algorithm in the simulation environment. A robot is instantiated in USARSim using MOAST and provided with a goal point to drive to. Using the ServoGUI tool in MOAST, a bounded region which is defined by the path segment and the arc tolerances verse the actual path that was driven by the vehicle may be plotted. This is shown in Figure 5. To gather quantifiable measurements of performance, the deviation from the actual vehicle path from the bounded nominal path region may be calculated. The tolerance of the bounded nominal path region may be altered to obtain finer granularity int the performance measurements, depicted in Figure 5b. This measure of deviation, given a path segments and arc tolerances, is a quantifiable means of measuring and comparing the performance of the new Primitive Mobility BG module.

6. CONCLUSION

In this paper, the importance of obtaining quantifiable means to measure the performance of algorithms for autonomous systems in a repeatable and comparable manner is understood. Section 4 provided an overview of how the combination of MOAST and USARSim can used to provide a baseline simulation system that is capable of modeling the embodiment and intelligence of an autonomous system, and the means to conduct repeatable experiments that target specific aspects of the autonomous system. Section 5 illustrates a methodology for utilizing the tools and facilities packaged with USARSim and MOAST to develop and test an algorithm for an autonomous system and how the systems can be used to obtain quantifiable measurements of performance. This baseline simulation system provides the basis to obtain results that can be compared and would enable developers to monitor the progress during the development of algorithms for autonomous systems. In conclusion, it has been shown how developers can utilize the combination of MOAST and USARSim to develop, test, and analyze algorithms for autonomous systems.

7. FURTHER WORK

There are several areas of active research and development in USARSim and MOAST. In USARSim, new sensor and manipulator models are being developed. New worlds are being developed that provide varying degrees of

difficulty in the simulation environment in regards to mobility and sensory perception. In MOAST, algorithms for the upper echelons of the hierarchy are being researched, which would implement tactical behaviors for a team of robots. However, the authors believe that the expansion of the knowledge repositories will yield the most benefit. This would enable developers to study the knowledge requirements needed to develop robust algorithms and would help guide researchers to find novel ways of obtaining the knowledge needed for rational decision making in autonomous system.

8. DISCLAIMER

Certain commercial software and tools are identified in this paper in order to explain our research. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the software tools identified are necessarily the best available for the purpose.

REFERENCES

1. M. Lewis, k. Sycara, and I. Nourbakhsh, "Developing a Testbed for Studying Human-Robot Interaction in Urban Search and Rescue," in *Proceedings of the 10th International Conference on Human Computer Interaction*, *HCII'03*, pp. 270–274, 2003.
2. S. Russell and P. Norvig, *Artificial Inelligence: A Modern Approach*, Prentice-Hall, 1995.
3. C. Thorpe, M. Hebert, T. Kanade, and S. Shafer, "Toward Autonomous Driving: The CMU NavLab. I - Perception," *Expert, IEEE [see also IEEE Intelligent Systems]* , 1991.
4. K. Dixon, J. Dolan, W. Huang, C. Paredis, and P. Khosla, "Rave: A Real and Virtual Environment for Multiple Mobile Robot Systems," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'99)*, **3**, pp. 1360–1367, October 1999.
5. D. Blank, H. Yanco, D. Kumar, and L. Meeden, "Avoiding the Karel-the-Robot Paradox: A Framework for Making Sophisticated Robotics Accessible," in *Proceedings of the 2004 AAAI Spring Symposium on Accessible, Hands-on AI and Robotics Education*, 2004.
6. G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, A. N. Marshall, A. Scholer, and S. Tejada, "Gamebots: A Flexible Test Bed for Multiagent Team Research," *Communications of the ACM* **45**, pp. 43–45, January 2002.
7. B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems," in *Proceedings of the International Conference on Advanced Robotics*, *ICRA 2003*, pp. 317–323, 2003.
8. R. Vaughan, B. Gerkey, and A. Howard, "On device abstractions for portable, reusable robot code," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, *IROS 2003*, 2003.
9. J. Albus, H. Huang, E. Messina, K. Murphy, M. Juberts, A. Lacaze, S. Balakirsky, M. Shneier, T. Hong, H. Scott, F. Proctor, W. Shackleford, J. Michaloski, A. Wavering, T. Kramer, N. Dagalakis, W. Rippey, K. Stouffer, S. Legowik, R. Bostleman, R. Norcross, A. Jacoff, S. Szabo, J. Falco, B. Bunch, J. Gilsinn, T. Chang, A. Meystel, A. Barbera, M. Fitzgerald, M. DelGiorno, and R. Finkelstein, "4D/RCS Version 2.0: A Reference Model Architecture for Unmanned Vehicle Systems," NISTIR 6910, NIST, 2002.
10. R. Adobbati, A. Marshall, A. Scholer, S. Tejada, G. Kaminka, S. Schaffer, and C. Sollitto, "Gamebots: A 3d Virtual World Test-Bed for Multi-Agent Research," in *Proceeding of the 2nd Workshop on Infrastructure for Agents, MAS, and Scalable MAS at Autonomous Agents*, 2001.
11. SourceForge.net, "MOAST," 2006. *http://sourceforge.net/projects/moast.*
12. SourceForge.net, "USARSim," 2006.*http://sourceforge.net/projects/usarsim.*
13. V. Gazi, M. Moore, K. Passino, and W. Shackleford, F. Proctor, *"The RCS Handbook: Tools for Real-Time Control Systems Software Development,"* Wiley Series on Intelligent Systems, John Wiley and Sons, Inc., 2001.
14. G. Miller, C. Fellbaum, R. Tengi, S. Wolff, P. Wakefield, H. Langone, and B. Haskell, "Wordnet Search: A Lexical Database for the English Language," 2006.

15. J. Albus and A. Meystel, *Engineering of Mind: An Introduction to the Science of Intelligent Systems*, John Wiley and Sons, Inc 2001, 2001.
16. Physics Laboratory of NIST, "International System of Units (SI)," 2000. *http://physics.nist.gov/cuu/Units/index.html.*
17. Society of Automotive Engineers, "Vehicle Dynamics Terminology, SAE J670e," Tech. Rep. J670e, SAE, 1976.