A Mobile Robot Control Framework: From Simulation to Reality

Stephen Balakirsky, Frederick M. Proctor, Christopher J. Scrapper, and Thomas R. Kramer

National Institute of Standards and Technology Gaithersburg, MD USA 20899

Abstract. In order to expedite the research and development of robotic systems and foster development of novel robot configurations, it is essential to develop tools and standards that allow researchers to rapidly develop, communicate, and compare experimental results. This paper describes the Mobility Open Architecture Simulation and Tools Framework (MOAST). The MOAST framework is designed to aid in the development, testing, and analysis of robotic software by providing developers with a wide range of open source robotic algorithms and interfaces. The framework provides a physics-based virtual development environment for initial testing and allows for the seamless transition of algorithms to real hardware. This paper details the design approach, software architecture and module-to-module interfaces.

1 Introduction and Related Work

The usefulness of simulation for developing control systems is well established. The role of simulation is to provide convincing sensor measurements in response to a controller's actuator outputs in an environment observable to developers. Ideally the simulation should be accurate enough so that performance parameters tuned in simulation work as well in the real world. In practice, attaining this level of simulation is often more costly than real-world testing, and simulators that respond plausibly if not accurately are acceptable. Plausible simulation then complements real-world testing to minimize the time and effort needed to build controllers that work well. Several such simulation systems exist; several of which are open source, including the Unified System for Automation and Robot Simulation (USARSim) [1] [1] and the Stage and Gazebo components of Player/Stage [2].

While the typical simulation system allows one to directly connect and experiment with servo-level controllers, they in general lack any form of intelligence

[1] No approval or endorsement of any commercial product by the National Institute of Standards and Technology is intended or implied. Certain commercial equipment, instruments, or materials are identified in this report in order to facilitate understanding. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

or ability to interpret sensor readings and issue meaningful commands. For this reason, it is necessary to connect the simulation engine to a control framework. Several such systems exist in the literature and on the web. Perhaps the most popular of which is the Player portion of Player/Stage.

Player/Stage combines a robot server interface, called Player, with a simulation system, called Stage, so that Player-enabled robots can be easily interchanged with each other and their simulated counterparts. The Player interface is installed on robotic vehicles, providing an interface to the robot's sensors and actuators over a TCP/IP network. Stage simulates a population of robotic vehicles and sensors in a 2-D environment. Gazebo is a 3-D counterpart provided for outdoor simulation. While Player started as a robot interface with drivers that directly control hardware, it has grown to include several abstract drivers since then. These abstract drivers use other drivers, instead of hardware, as the sources for data and the sinks for commands. Several well-known algorithms are now included with the system thus providing services such as way-point navigation and obstacle avoidance.

Several component-based architectures have been developed. These include the middleware project RT-Middleware [3] and the component architecture OCRA [4]. These architectures provide a component specification that prescribes a component's interfaces, activity, and input/output ports. They do not provide a functional architecture. In the case of RT-Middleware, a graphical tool may be used to interconnect various components and create a fully functional robotic system.

USARSim provides robot and sensor models and a standardized actuator/raw-sensor level interface for communicating with a physics-based simulation engine. Many of the robots and sensors have been validated against their real counterparts. In addition, USARSim supports a Player driver that allows algorithms coded to its interface specification to utilize Player to communicate with real hardware.

A new entry to the robot simulation/control arena is Microsoft Robotics Developer Studio (MSRDS) [5]. MSRDS includes support for simulation and implements services to control robotic platforms. While MSRDS is not as mature as Player/Stage or USARSim, it promises to build a library of services that will be available to robot developers.

1.1 Adding an Architecture: MOAST

Player, USARSim, and MSRDS focus on the interfaces to mobile robots that allow developers to build their own controllers, with portability across robots that support Player or MSRDS made easier. Rt-Middleware and OCRA provide a component level specification. None of the systems defines an overall architecture or high-level interface specification that guides the development of robot controllers. We have found that an architecture is essential to the efficient development of intelligent systems. An architecture assigns roles and responsibilities among controllers and dictates what services are necessary. It defines module timing, data and control interfaces, and planning extents. An architecture also

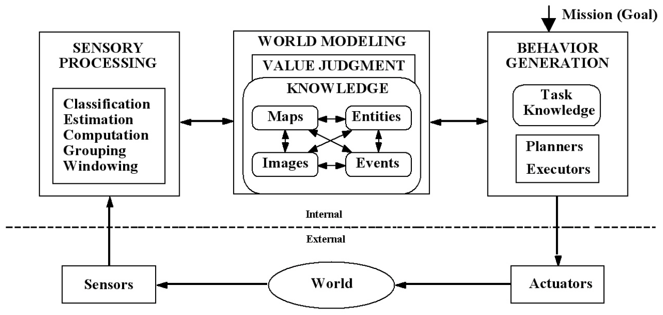

Fig. 1. Generic 4D/RCS Control Node.

provides the framework in which the rest of the intelligent system resides and dictates the rules that the modules must follow. For these reasons, we built the Mobility Open Architecture Simulation and Tools Framework (MOAST). MOAST begins with a well defined architecture, and adds simulations, services, and controllers. The entire MOAST framework is intended to provide tools to lead a researcher through all of the phases of development and testing of an autonomous agent system.

MOAST is made up of the following components:

1. A reference model architecture that dictates how control responsibilities are divided between modules.
2. Communication interface specifications that dictate how and what modules will communicate.
3. Sample control modules for the control of a sample simulated robotic platform. These modules include sensor processing, world modeling, and behavior generation for 4 levels of the hierarchical architecture and provide a complete control system.
4. Validated sensors and robot models in the simulation.
5. Tools to aid in development and debug of the control system.

The remainder of this paper will address the components of MOAST. Section 2 describes the reference model architecture that is utilized by MOAST. Section 3 describes the various services and capabilities that are provided by the framework, Section 4 describes how MOAST transitions from simulation to real hardware. Finally, Section 5 describes future work and concludes the paper.

2 Reference Model Architecture

The capabilities of the MOAST framework are encapsulated in components that are designed based on the 4D/RCS Reference Model Architecture [6][7]. The RCS reference model architecture is a hierarchical, distributed, real-time control system architecture that decomposes a robotic system into manageable pieces while providing clear interfaces and roles for a variety of functional elements.

Figure 1 depicts the general structure of each echelon (level) of the 4D/RCS hierarchy. Each echelon in 4D/RCS contains a systematic regularity and is composed of control nodes that 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). WM must store this information, information about the system self, and general world knowledge and rules. Furthermore, it must provide a means of interpreting and accessing this data. BG computes possible courses of action to take based on the knowledge in the WM, the system's goals, and the results of plan simulations. VJ aids in the BG process by providing a cost/benefit ratio for possible actions and world states.

The principal difference between control nodes at the same echelon is in the set of resources managed, while the principal difference between nodes at different echelons is in the knowledge requirements and the fidelity of the planning space.

This regularity in the structure enables flexibility in the system architecture that allows scaling of the system to any arbitrary size or level of complexity [8].

2.1 Generic Module

While 4D/RCS provides a reference model for the architecture, MOAST is an implementation of that architecture. Therefore, specific responsibilities, knowledge requirements, and interfaces have been designed for each control module. Each control module is based upon a generic core controller that is shown in Figure 2. The MOAST hierarchical decomposition in terms of its control modules is depicted in Figure 3.

The control module core has the following flow:

1. **Initialize:** The initialization opens any communication buffers, places the system in a safe known state, and initializes any control parameters.
2. **Read Command:** Command information is received, and the system is prepared to execute the command. When a new command is received, the old command is immediately replaced by this command.
3. **Read Config:** Configuration information is received, and the system is prepared to change its settings. A separate configuration channel is provided to allow for control parameters to be changed without interrupting the current controller. For example, a user may want to change a system's cycle time without interrupting a complex control function.
4. **Handle Command State Tables:** All of the modules run on a fixed cycle time. Therefore, command functions must either guarantee that they finish in under the cycle time, or provide for being re-entrant. Although this text refers to the command execution as being finite state machine (FSM)

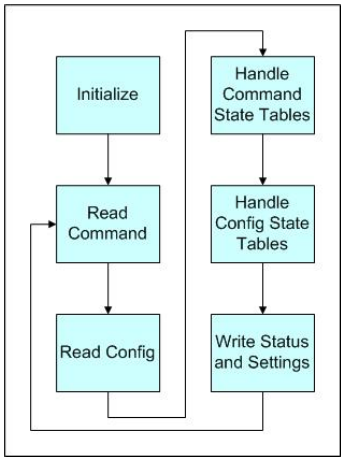
Initialize
Read Command
Read Config
Handle Command State Tables
Handle Config State Tables
Write Status and Settings

Fig. 2. Generic core of control module.

based, search based planning systems have also been implemented under this framework.

5. **Handle Config State Tables:** Requests to change configuration settings are carried out similarly to handling command state tables.
6. **Write Status and Settings:** The current module status and the configuration's settings are sent out over communication buffers.

2.2 RCS Library

Support for developing software conforming to the 4D/RCS methodology is provided by the RCS Library [9]. The RCS library includes portable utilities for creating and synchronizing real-time tasks following the 4D/RCS architecture. Code generation and diagnostics tools simplify much of the application setup and debugging. Communication between RCS control modules is provided by the Neutral Message Language (NML), a software library for communication ported to a variety of platforms including Linux, Solaris, VxWorks, LynxOS, QNX, Windows and MacOS. Applications using NML define a message vocabulary as C++ classes and call C++ methods to open buffers, read and write messages. Java bindings are also available. NML applications running on one platform can communicate with ones running on any other platform. The location of buffers and processes that connect to them is selected at run time, and a running application can be extended to communicate with new processes dynamically. NML source code is freely available at [10] and documented in [9].

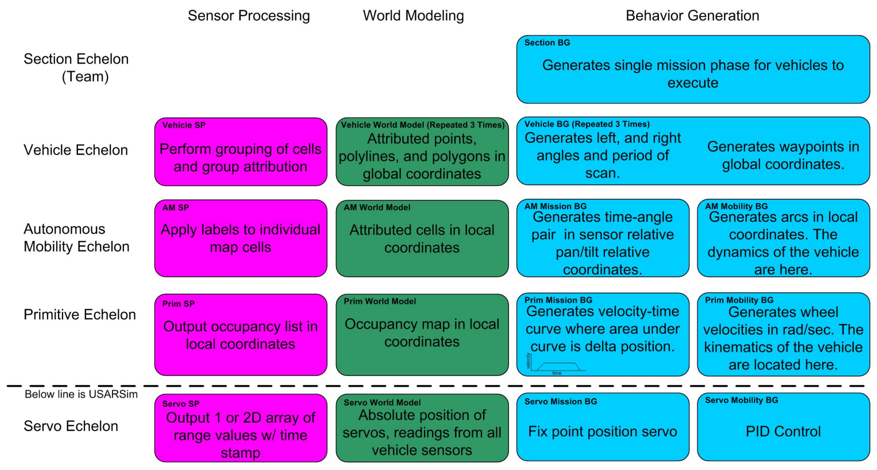
Sensor Processing
World Modeling
Behavior Generation
Section Echelon (Team)
Section BG
Generates single mission phase for vehicles to execute
Vehicle Echelon
Vehicle SP
Perform grouping of cells and group attribution
Vehicle World Model (Repeated 3 Times)
Attributed points, polylines, and polygons in global coordinates
Vehicle BG (Repeated 3 Times)
Generates left, and right angles and period of scan.
Generates waypoints in global coordinates.
Autonomous Mobility Echelon
AM SP
Apply labels to individual map cells
AM World Model
Attributed cells in local coordinates
AM Mission BG
Generates time-angle pair in sensor relative pan/tilt relative coordinates.
AM Mobility BG
Generates arcs in local coordinates. The dynamics of the vehicle are here.
Primitive Echelon
Prim SP
Output occupancy list in local coordinates
Prim World Model
Occupancy map in local coordinates
Prim Mission BG
Generates velocity-time curve where area under curve is delta position.
Prim Mobility BG
Generates wheel velocities in rad/sec. The kinematics of the vehicle are located here.
Below line is USARSim
Servo Echelon
Servo SP
Output 1 or 2D array of range values w/ time stamp
Servo World Model
Absolute position of servos, readings from all vehicle sensors
Servo Mission BG
Fix point position servo
Servo Mobility BG
PID Control

Fig. 3. Modular decomposition of MOAST framework that provides modularity in broad task scope and time.

3 MOAST Provided Functions

The development and maintenance of an advanced mobile robot require expertise ranging from sensor processing, path planning, and communications protocols, to basic auto repair. While many of the algorithms for accomplishing these functions are well known, freely available code that implements these functions tends to be incompatible with other code or robotic platforms. This necessitates interface and functional tweaks before the code modules become useful.

Part of the original design philosophy of MOAST was to provide “out of the box” functionality that would reduce the breadth of expertise required to conduct research with mobile robots. The developers of MOAST have taken many well known algorithms and implemented them within the 4D/RCS framework. The result is a fully functional framework that allows researchers and students to immediately begin to experiment with functional robots in both simulated and real environments. Researchers are then free to examine the code modules that address functions in their areas of expertise. The hope is that as improvements are made, the researchers will contribute the improved modules back to the community. The basic functionality of the mobile robot may be broken down in the the areas of sensor processing and mobility.

3.1 Sensor Processing

The majority of the sensor processing work performed in MOAST is in the detection of obstacles. The decomposition of this responsibility by echelon in the MOAST framework is shown in Table 1. For the laser scanner, the Primitive (Prim) Echelon provides a series of data tuples as shown in Figure 4. The data

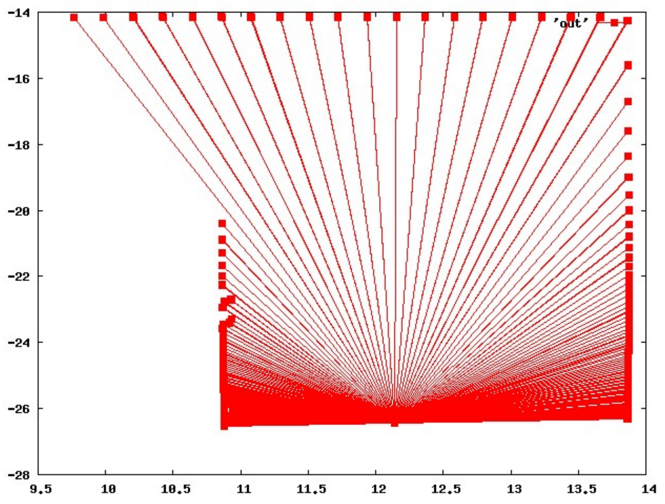

Fig. 4. Laser range data from Prim includes the start and end of each beam.

Table 1. Sensor processing requirements and responsibilities.

Data Out	Description
Primitive Echelon laser scan data	Beam start and hit point
Autonomous Mobility Echelon height map	Cellular map of 2.5D elevations
Autonomous Mobility Echelon obstacle probability	Cellular map of obstacle probabilities
Vehicle Echelon obstacle map	Concave hull of obstacle areas from AM

is available over the communications interface and includes the location of the device when the beam was fired and the beam hit point in vehicle relative coordinates. Under the current system, the laser is constrained to be fixed mounted and facing straight ahead of the vehicle in a level orientation. While this presents an instantaneous snapshot of the environment, the data tends to be noisy, and encompass a very small region. This data is further processed to produce a cellular height map of the environment as shown in Figure 5. Due to the mounting constraint on the laser, whenever the vehicle is driving on a flat level surface the height of every cell that has been observed is either the height of the floor or the height of the laser above the floor.

While the external representation is transmitted as a cellular height map, internally, the cell's height, range, hits history, and obstacle probability are stored. The model for the terrain being observed is like a 3D bar chart, where solid blocks of various heights extend through cells in the XY plane. The height of each cell records the estimated distance its block extends above the local XY plane. The

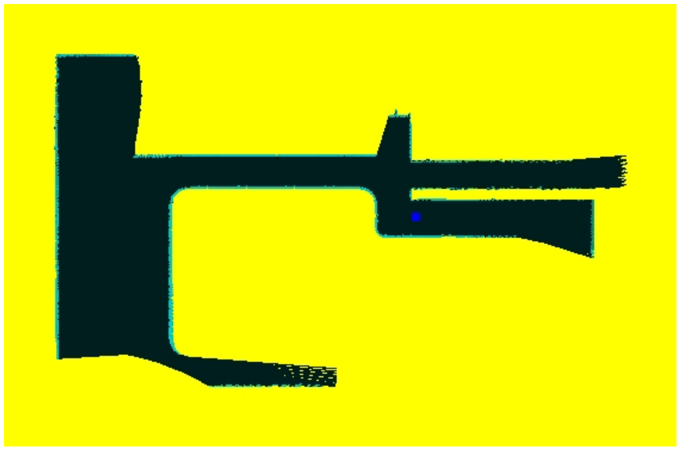

Fig. 5. Cellular height map generated from laser data. Yellow represents cells that have never been seen, and cells that are observed are shown in shades of aqua based on their height. Due to the mounting configuration of the laser, only heights of "ground," shown as very dark aqua (i.e. black), or heights of above the laser, shown as bright aqua, are displayed (hard copies of this paper should be printed in color).

height is negative if the top of the block is below the XY plane. The range of a cell records the largest distance from which a cell has been seen to contain an obstacle. Some obstacles are seen only when they are close to the sensor. It is desired to avoid having the system decide that an obstacle no longer exists because it is not seen when the system is farther away than its range. It is expected that if a cell containing an obstacle is viewed from within the cell's range, the obstacle will be seen again, but if the cell is viewed from beyond its range, the obstacle might not be seen. The range is used in setting the hits history, as described below.

The hits for a cell encodes the seven most recent viewings of the cell. A cell is regarded has having been viewed whenever a ladar ray passes through it (the cell is not seen) or bounces off an obstacle in it (the cell is seen).

Obstacle probability is a real number from 0.0 to 1.0. It represents the system's best estimate of the chances that the cell is occupied by an obstacle. A separate map of obstacle probability is exported over a communication channel for use by other modules.

3.2 Mobility

The mobility functions consist of a family of planning algorithms that are able to compute obstacle free paths for Ackerman, skid-steered, and omni-drive ground robot platforms as well as helicopter-like air platforms and sub-like underwater vehicles. When examining the planning systems, it is useful to note the knowledge

Table 2. Mobility planning requirements and responsibilities.

Plan Out	Command In	Knowledge In
Prim actuator/motor commands	Constant curvature arcs	Kinematics
AM constant curvature arcs	Way-points	Dynamics
Vehicle way-points	Named location	*a priori* map
Section vehicle actions	Behaviors	Vehicle Capabilities

required by each module as well as the module's output format, i.e., the form the plan takes. This information is represented in Table 2.

At the lowest echelon, the output of the planning system consists of actuator and motor commands that are sent through MOAST's generic interface known as SIMware [11]. These commands are platform steering type dependent and consist of such things as left and right wheel velocities for a skid-steered vehicle or steering curvature and velocity for an Ackerman steered vehicle. This module requires a series of obstacle free constant curvature arcs as input. In addition to the command input, the module requires knowledge of the specific robot kinematics. Item such as wheelbase, tire diameter, and minimum turning radius must be provided.

An additional way-point interface exists into the planning hierarchy. This interface accepts a series of way-points as its commands and computes a series of obstacle-free constant curvature arcs as output. This module reads in the obstacle probability map from the sensor processing chain and also has knowledge of the vehicle dynamics. A graphical example of this module's output is shown in Figure 6.

This planning module has two main strengths. It quickly plans realistic smooth paths with appropriate speeds and curve radii while keeping within the allowed deviation and avoiding obstacles. Second, it plans paths dynamically in environments with moving obstacles (such as other vehicles). Weaknesses of this planner stem primarily from not getting enough sensory information and not attempting to use all the information available.

If *a priori* data is available, then a planning module exists to take advantage of this data. This module ingests *a priori* vector data and computes a visibility graph based plan that starts at the way-point planner's planning horizon and terminates at a named point (for example an address). This system currently reads .mif formatted vector data. An example of the plan output is shown in Figure 7. The system accepts a named point as its input and outputs a list of way-points for the platform to follow.

Finally, a planning system capable of coordinating groups of vehicles exists. This planner accepts behavior based commands (i.e. explore, or deliver packages) and coordinates the actions of several platforms to accomplish the tasks. The system accepts the behavior command as its input and outputs named points and tasks for the platforms to accomplish. The system must have knowledge of individual platform capabilities.

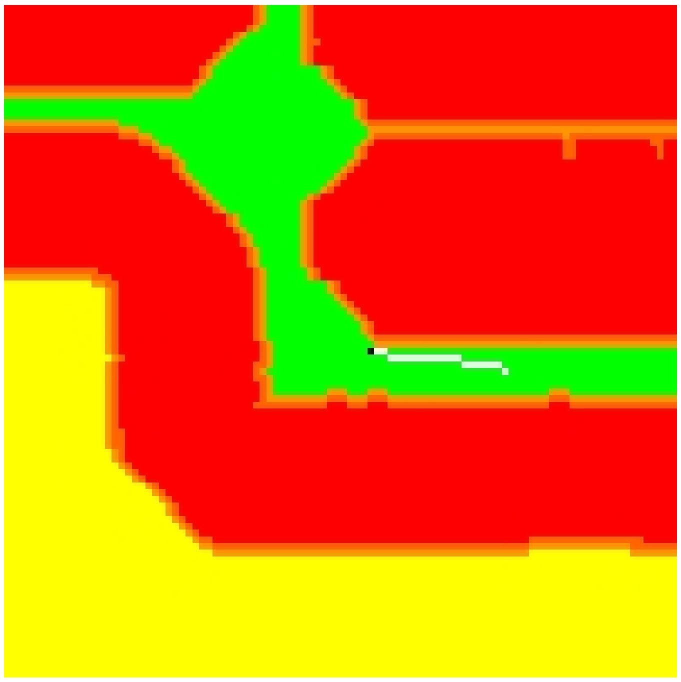

Fig. 6. Cellular obstacle map generated from obstacle probability data. Yellow represents cells that are unknown. Green represents free-space, orange represents the edge of obstacles, and red represents obstacles. The obstacles are grown by half the vehicle width to allow for the planner to plan on a point-sized robot. The white path represents the planned path for the platform and the platform's current location is shown as the black dot (hard copies of this paper should be printed in color).

4 Migration to Real Hardware

The Servo Echelon in Figure 3 is implemented outside of MOAST by real or simulated vehicles. To limit the spread of vehicle-specific source code into MOAST, an external middleware layer was built that bridges different controllers to different vehicles or vehicle simulations. This Simulation Interface Middleware (SIMware) defines a software application programming interface (API) based on the notion of "skins" that customize an environment to particular controllers and simulations or real vehicles [11]. Skins are divided into superior skins that interface SIMware to vehicle controllers, and inferior skins that interface SIMware to simulations or real vehicles. Programmers build a SIMware middleware layer by instantiating a particular superior skin that interfaces to a controller, instantiating a particular inferior skin that interfaces to a robot and sensor environment, and defining configuration information specific to each skin.

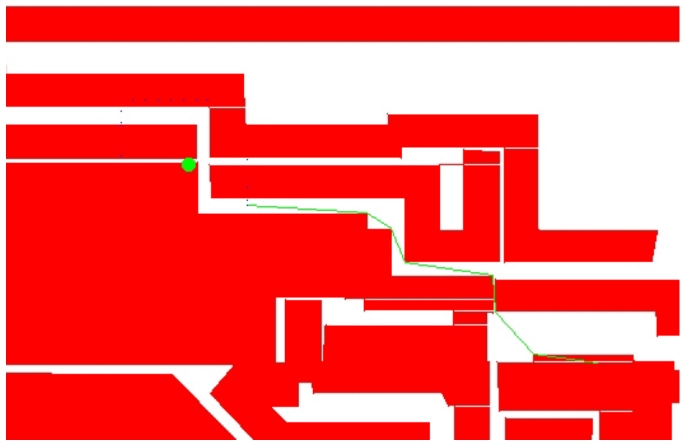

Fig. 7. Vector-based *a priori* map used by the *a priori* planner. The red areas are obstacle polygons and the white is free space. The computed path is shown in green, and the robot location is shown with a large green circle. The small green dots represent the planning horizon of the way-point planning system.

Inferior skins have been created that communicate with the simulator USARSim, the Player interface library, and directly to smart motor drives. This has allowed for the direct application of simulated code to real platforms. Since all of the interfaces above the Servo Echelon do not change, no modifications to the algorithms under test are necessary. In fact, by using a validated simulation engine such as USARSim, the authors have been able to migrate the entire MOAST framework from simulation to an ATRV platform without changing any lines of code. In addition, real/virtual operation is possible where part of the system operates in simulation while other aspects are run on real hardware. This has been demonstrated with the use of real mobility and simulated perception. In this case, mobility planning algorithms are able to take advantage of perceived attributes that may not be available from the current generation of perception algorithms.

5 Future Work and Conclusions

The MOAST framework has been used to control virtual robots in both urban search and rescue environments and manufacturing settings, and physical robots (automated guided vehicles) on real shop floors. By utilizing the Player inferior skin of SimWare, identical algorithms that have been tuned in simulation are being experimented with on real hardware in identical environments. The idea is to validate performance in both the real and virtual worlds in order to verify simulated models and control system utility.

In addition, new algorithms are constantly being added to the framework. Work is progressing on Simultaneous Localization and Mapping (SLAM) as well as the inclusion of a true 3D world model. The MOAST website highlights the latest improvements.

References

1. USARSim. `www.sourceforge.net/projects/usarsim`.
2. The Player Project. `playerstage.sourceforge.net`.
3. Ando, N., Suehiro, T., Kitagaki, K., Kotoku, T., and Woo-Keun, Y., 2005. "Rt-middleware:distributed component middleware for rt (robot techonlogy)". In Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp. 3933–3938.
4. Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Oreback, A., 2005. "Towards component-based robotics". In Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp. 163–168.
5. Microsoft Robotics Studio. `msdn.microsoft.com/robotics`.
6. Albus, J., 1991. "Outline for a theory of intelligence". *IEEE Transactions on Systems Man and Cybernetics,* **21**, pp. 473–509.
7. Albus, J., 2000. "4D/RCS reference model architecture for unmanned ground vehicles". In Proceedings IEEE International Conference on Robotics and Automation, pp. 3260– 3265.
8. Balakirsky, S., and Scrapper, C., 2004. "Planning for on-road driving through incrementally created graphs". In Proceedings IEEE Conference on Intelligent Transportation Systems.
9. Gazi, V., Moore, M. L., Passino, K. M., Shackleford, W. P., Proctor, F. M., and Albus, J. S., 2001. *"The RCS Handbook: Tools for Real-Time Control Systems Software Development"*. John Wiley and Sons.
10. The Real-Time Control Systems Library. `www.isd.mel.nist.gov/projects/rcslib`.
11. Scrapper, C. J., Proctor, F. M., and Balakirsky, S., 2007. "A simulation interface for integrating real-time vehicle control with game engines". In Proceedings of the ASME Computers in Engineering Conference, September 3-7 2007, Las Vegas, Nevada USA, pp. DETC2007–34495.