Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

THE MITRE METEOR ROBOT CONTROL SOFTWARE: SIMULATE AS YOU OPERATE

Richard M. Weatherly
Frederick S. Kuhl
Robert H. Bolling
Robert J. Grabowski

The MITRE Corporation
7515 Colshire Drive
McLean, VA 22102-7508, U.S.A.

ABSTRACT

The Defense Advanced Research Projects Agency (DARPA) challenged autonomous ground vehicle developers in the "2005 DARPA Grand Challenge" to build a vehicle that could complete a 132 mile course through the American desert southwest. MITRE, a not-for-profit systems engineering company, responded to this challenge by creating the MITRE Meteor in just 11 months. This rapid development relied on software employment transparency to get the maximum utility out of each line of code. Judicious design of the software framework allowed the same body of code to animate the robot in the field, support laboratory experimentation, and analyze recorded field testing data. This paper describes how software employment transparency was achieved and how it increased development efficiency.

1 INTRODUCTION

The Defense Advanced Research Projects Agency (DARPA) reissued a challenge to developers of autonomous ground vehicles in 2005 to build machines that could complete a 132 mile, off-road course. Of the 195 initial entrants only 23 qualified to compete in the race. Selection was based on several down selects including 10 days of rigorous competition at the National Qualifying Event held at the California Speedway. The final race was a few days later on October 8th and 9th in the Mojave Desert. The course included gravel roads, dirt paths, switchbacks, open desert, dry lakebeds, mountain passes, and two tunnels. The vehicle needed to navigate GPS waypoints on a prescribed course while staying within defined boundaries and avoiding obstacles including other robotic vehicles. The route was given to the teams only two hours before the race began.

The MITRE Corporation decided to participate in the Grand Challenge in late September 2004 (see Figure 1). MITRE, the primary sponsor for the MITRE Meteor team, invested discretionary funds in the event believing that MITRE's work programs and sponsors would benefit from an investigation of the technologies that contribute to the DARPA Grand Challenge.

MITRE's team had no previous experience with the Grand Challenge and was operating under a very difficult time constraint. This led to several team philosophies. First, employ commercial solutions wherever possible. Stated another way, apply the main focus to the areas that need the most innovation and farm out the rest. Second, use an incremental model-simulate-test approach. Build a model that is suitable to the current task. Verify and tune the model using simulation and replay. Test the model and system in real situations and then use the results of the testing to adjust the model as necessary. This approach promotes increasing sophistication commensurate with the current capabilities of the robot while respecting the ultimate goal. Third, use employment transparency to get the maximum utility from our software development investment. That is, craft the control software and execution environment so that it can be used for more than one purpose.

Figure 1: MITRE Meteor at the Finals of the 2005 DARPA Grand Challenge Robot Race

2 SOFTWARE EMPLOYMENT PHASES

The Meteor software development team had three jobs: build real-time code that would drive a 5000 pound truck down the road without hurting anyone, support robot control algorithm research, and construct tools to analyze operational data recorded in the field. Given the tight schedule, 11 months, it was clear we could not afford the time and coordination overhead required to manage these three jobs as independent efforts. Our goal was, therefore, to create a single body of code that could be employed in the operational, developmental, and analytical phases of our project with minimum modification.

2.1 Operational Employment

The Meteor control software is a group of message passing agents, as shown in Figure 2. The population of agents processes sensor input and produces vehicle control commands. Each agent has its own thread of control implemented as a Java thread and mapped to a supporting OS level thread. There are three types of agents: lookouts, watch officers, and executives. The type distinction is based on the control flow pattern of the basic loop within the agent. Lookouts perform I/O with physical sensors and send messages containing raw sensor data to the balance of the agent population. Watch officers receive messages, perform some value-added processing (typically sensor fusion), and report their results in messages to the rest of the agents. Executives assess the system state, perform some value-added computation, report their results through messages, and then wait for some amount of time to pass.

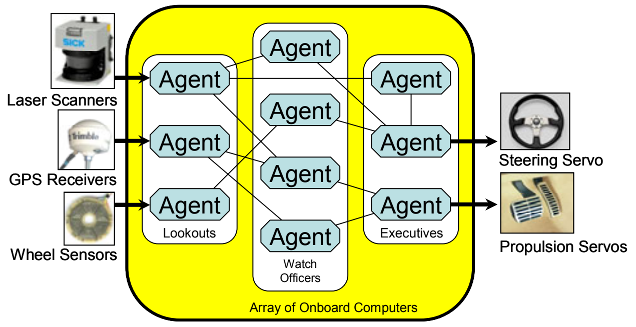

Figure 2: Operational Employment

Another way to view the agent type distinction is by the event that triggers agent action. Action in lookouts is triggered by the reception of physical I/O. Watch officers take action when they receive a message, while executives act when some amount of time elapses (typically fixed deltas resulting in 10Hz to 80Hz loops).

When installed on the robot, the agents are distributed across an array of 7 processors. The processors are connected by a 1 gigabit Ethernet bus and run Fedora Core 3 and the Sun Java Runtime Environment. The large number of processors in the array was a decision taken early in the project as a hedge against unknown processing demand. As it tuned out, the final compliment of agents could run on as few as three processors.

2.2 Development Employment

Figure 3 illustrates the software configuration used for the algorithmic development of the Meteor agents. A simple vehicle motion model was constructed that transforms steering and propulsion commands, as they would have been sent to the servo controller digital inputs, into vehicle location, orientation, and velocity values. These values are supplied to three sensor models.

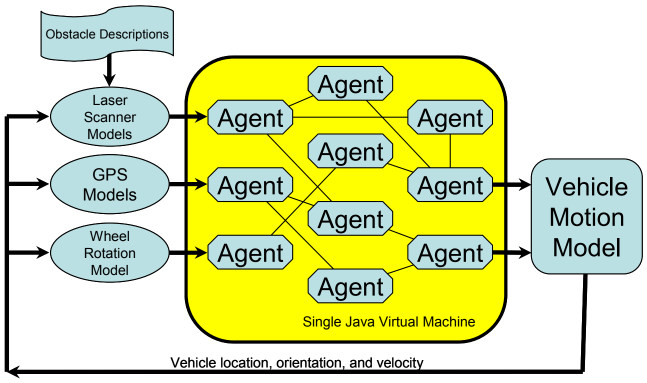

Figure 3: Quasi-Real Time Simulation Employment

The laser sensor model takes the changing vehicle location and orientation values and generates a stream of laser output frame messages. These frames are identical in format to those produced by the actual physical laser sensor. The model does this by considering the relative geometry of the moving vehicle versus a static collection of obstacle descriptions. The set of obstacle descriptions was handmade and roughly corresponds to the buildings and trees surrounding a parking lot near MITRE’s McLean Virginia office.

The GPS models take the location, orientation, and speed of the vehicle, as computed by the vehicle motion model, adds statistical noise to the values, and then report them in messages. These messages are identical to those reported by the physical sensors and include fabricated values for various GPS signal quality metrics.

The wheel rotation model takes the stream of changing vehicle location values from the vehicle motion model and transforms them into the displacement, speed, and acceleration values that are physically generated by an embedded processor on the robot that measures drive shaft rotation.

The entire software configuration can be run on a single workstation. This configuration was used with great benefit as the soft-ware system design first evolved. The fact that the system only ran in real time was not a problem except when trying to evaluate vehicle behavior over long periods of time. This was particularly taxing when simulating vehicle performance for the full 10 hours allowed by DARPA to complete the race.

In addition to the real time limitation, results were often not reproducible. The primary reason for this is competition for limited computing power. An assumption of the agent-based architecture is that all executive agents have enough computational power available to complete the processing of one cycle before it is time to begin the next cycle. This assumption is seldom violated when the agents run on the robot's powerful, multi-processor array and is constantly violated when they run on a single workstation. When an executive agent does not finish one cycle before it is time to begin the next, controller frequencies become random and the simulation ceases to reflect the real system.

To improve reproducibility and support both faster and slower than real time simulation, the Tortuga discrete event simulation framework (Weatherly and Page 2004) was incorporated, Figure 4. This required three changes. First, the agent framework was extended to include a mode switch. In one switch position (operational mode), each agent is a real OS thread and the population of agents operate as they would when installed on the vehicle processor array. In the other position (simulation mode), agents become interacting sequential logical processes in the Tortuga framework.

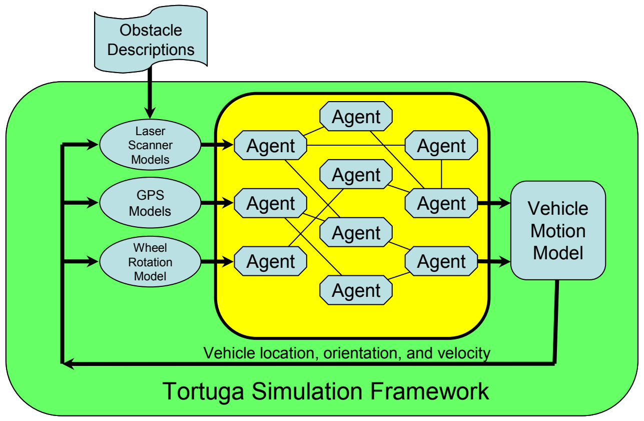

Figure 4: Discrete Event Simulation Employment

The second change required to introduce Tortuga involves delays. Attempts by an agent to delay its execution are intercepted. How the delay request is interpreted was made mode switch dependent. In operational mode, an attempt to pause *t* milliseconds results in a call to `Thread.sleep(t)`. In simulation mode, a similar attempt results in the suspension of the agents logical process and the scheduling of a resumption event *t* milliseconds of simulation time in the future.

The third change for Tortuga involves messages. The inter-agent communication system was made mode switch dependent. Specifically, a minor change was made to each agent's input message queue. In operational mode, the input queue is a Java `LinkedBlockingQueue` that blocks agent threads that attempt to read from an empty queue. In simulation mode, input messages are simply scheduled events that resume the agent's logical process when their time arrives.

2.3 Analytical Employment

An advantage of the agent-based architecture is access to the evolving computational state. A good view of how the agents are working internally and behaving as a group can be obtained from recording their message exchange. Such recordings were made each time the robot was operated in the field and archived for later study. This was important given the limited opportunity the team had for actual autonomous vehicle operation before the race. There are few places in the suburbs of Washington DC where it is safe to let an autonomous pick-up truck run free.

Figure 5 shows how recorded messages are analyzed. Typically, the team returned from the field wanting an answer to the recurring question; "Why on earth did it do that?" To explore field testing events, two things were done. First, a small special purpose agent called the Replayer was built that could read messages that were recorded in the field and then introduce them into the Tortuga framework as scheduled events. Second, agents that manage sensors were removed from the population. With such an arrangement, recorded messages take the place of real sensors. The balance of the agents behave exactly as they do in the field. Using controls in Tortuga that set the ratio of simulation time to real time, the analyst can fast-forward though the recorded messages until the time of the curious robot behavior. Then the analyst can slow the replay and attempt to determine what the robot was doing.

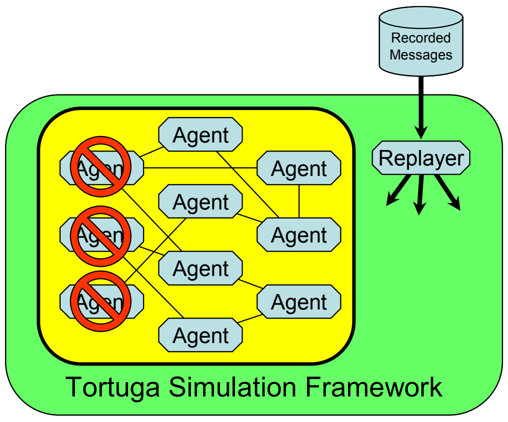

Figure 5: Replay of Recorded Messages

3 SIMULATION INFRASTRUCTURE

The success of the Meteor depends on the ability to run its control software either in operational or in simulation mode. The key to switching easily between these modes of employment is the infrastructure. As mentioned above, the control software "sees" the same interface regardless of the mode. The infrastructure allows the agents to function either as threads or as interacting-process simulation entities. The simulation infrastructure comes from the Tortuga framework.

3.1 Process Orientation and the Tortuga Framework

There are several views, or organizing principles, for discrete-event simulations. "The simulation structure that has the greatest intuitive appeal is the process-interaction method. The notion is that the computer program should emulate the flow of an object through the system. The entity moves as far as possible in the system until it is delayed, enters an activity, or exits from the system. When the entity's movement is halted, the clock advances to the time of the next movement of any entity. This flow, or movement, describes in sequence all the states that the object can attain in the system." (Banks 1998)

The Tortuga framework is a product of MITRE-sponsored research to facilitate the rapid construction of analysis and training simulations by small teams. Tortuga allows interacting-process simulations to be written in Java using current development tools and incorporating third-party and open-source software. In addition to the Meteor application, Tortuga has been applied successfully to novel simulations in air traffic control and military analysis.

Interacting-process simulation in Java is not new. There are several implementations (Jacobs et al. 2002, Gehlsen and Page 2001). These frameworks, owing to operating system limits, support only a few thousand logical processes. However, Tortuga supports simulations of hundreds of thousands of logical processes. Tortuga achieves its scalability through modifications to the Jikes Research Virtual Machine. These modifications enable efficient, lightweight coroutines. Tortuga also runs atop the Sun JVM with the same limitations as other frameworks on the number of logical processes; the Sun JVM underlies the work reported here.

Tortuga adds several features useful for military simulation, specifically action methods and triggers. An action method is a distinguished method on a simulation entity. When invoked by another entity, an action method has the side-effect of causing the simulation executive to schedule the invoked entity to resume execution. Thus the entity, which might have been waiting for simulation time to pass, is awakened beforehand. Action methods naturally represent the occurrence of exogenous events or interruptions, such as being shot at or the arrival of a message.

A trigger is a boolean predicate defined by the simulation writer (as a Java class with a method that evaluates the predicate). An entity can ask the executive to let it sleep until one or more triggers are satisfied or some amount of time passes. The analogy is with database triggers. Triggers allow an entity to wait for the state of the simulation to reach a certain condition, like other entities being with range of a sensor.

3.2 What Threads and Simulation Entities Have In Common

The Meteor control software is organized as agents. The central problem of supporting the various employment phases is supporting the agents in those phases. Agents need three things in each phase:

- A thread of control,
- A way to sleep for a set period of time, and
- A way to await the arrival of a message.

From the agent's perspective, the infrastructure software provides these three things to agents the same way in each phase.

3.2.1 A Thread of Control

In operational mode, each agent is animated by a Java Thread, that is, the agent's behavior, encapsulated in a method, is performed by a Thread. In simulation mode, the agent's behavior is performed by a Tortuga Entity, which runs as a logical process under control of the Tortuga simulation executive.

3.2.2 Sleeping For a Set Period of Time

Agents running in any employment mode sleep or delay for a set time by calling a method on a `TimeManager`. The `TimeManager` provided to an agent depends on the mode. In operational mode, the `TimeManager` provided merely performs a `Thread.sleep(t)`. In simulation mode, the `TimeManager` performs a Tortuga `waitForTime(t)` call, which suspends the underlying Entity's execution and returns control to the Tortuga executive.

3.2.3 Awaiting Arrival Of A Message

Agents await the arrival of a message by invoking `read()` on an abstract class Comm. In operational mode, the Comm instance is actually an instance of `OpComm`, which performs network communication. In simulation mode the Comm instance is actually an instance of `SimComm`. This class waits for a message to be placed in the agent's receive buffer. The agent, which in simulation mode is a Tortuga Entity, defines a Tortuga trigger that is satisfied by one or more messages waiting to be read. Thus the agent is suspended under control of the Tortuga executive until a message arrives in its buffer.

4 CONCLUSION

The Meteor software team achieved its goal of employment transparency using a single body of code for robot operation, code development, and logged data replay analysis. The software benefited greatly from its agent design, which allowed cooperative decomposition of its various reporting, analysis, and executive functions. Structuring and encapsulating cooperation between agents running in real time as Java Threads via messages made coordination straightforward.

The interacting-processes view of simulation affords another, similarly natural, way to coordinate cooperating agents or entities controlled by simulation time. This work demonstrated how the Meteor software capitalized on the similarities between thread-backed agents and simulation entities to facilitate its employment in very different modes.

ACKNOWLEDGMENTS

The authors wish to acknowledge the entire 2005 MITRE Meteor Team. It was a privilege to work with the energetic and innovative folks seen in Figure 6. From left to right: Frank Carr, Bob Bolling, Bob Grabowski, Richard Weatherly, Dave Smith, Ann Jones, Tiffani Horne, Mark Heslep, Keven Ring, Kevin Forbes, Mike Shadid, Laurel Riek, Alan Christiansen and Sarah O'Donnell (inset).

Figure 6: The 2005 MITRE Meteor Team at the California Motor Speedway

REFERENCES

Banks, J. 1998. *Handbook of Simulation*, Wiley.

Jacobs, P., N. Lang, and A. Verbraeck. 2002. DSOL: A Distributed Java Based Discrete Event Simulation Architecture. *Proceedings of the 2002 Winter Simulation Conference.* Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Gehlsen, B., and B. Page. 2001. A Framework For Distributed Simulation Optimization. *Proceedings of the 2001 Winter Simulation Conference.* Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

Weatherly, R. M., and E. H. Page. 2004. Efficient Process Interaction Simulation in Java: Implementing Co-Routines Within a Single Java Thread. *Proceedings of the 2004 Winter Simulation Conference.* Piscataway, New Jersey: Institute of Electrical and Electronics Engineers.

AUTHOR BIOGRAPHIES

RICHARD M. WEATHERLY led the design and construction of the Meteor infrastructure and control software. He received a Ph.D. in Electrical Engineering from Clemson University and is a Consulting Engineer with The MITRE Corporation. His e-mail address is <weather@mitre.org>.

FREDERICK S. KUHL designed and implemented the Tortuga framework with Weatherly. He holds the Ph.D. in Computer Science from Texas A&M University. He is a Senior Principal Engineer with The MITRE Corporation. His e-mail address is <fkuhl@mitre.org>.

ROBERT H. BOLLING leads the robotics department at the MITRE Corporation in McLean, Virginia, where he was responsible for the physical and electrical design integration of the Meteor. He is a retired Air Force Experimental Test Pilot and has a B.S. and M.E. in Electrical Engineering and M.S. in Aeronautical Science. His e-mail address is <rbolling@mitre.org>.

ROBERT J. GRABOWSKI is a lead engineer in the robotics department at the MITRE Corporation. He started his career as a Reactor Operator in the US Navy. He received his Ph.D. in Electrical Engineering from Carnegie Mellon University in 2004 with a focus on small robots. He now develops sensor and control algorithms for large, outdoor robots and robot teams with the MITRE team. His e-mail address is <rgrabowski@mitre.org>.