Software Requirements

Intelligent Ground Vehicle Competition (FIT-IGVC)

Members:
Brent Allard ballard2014@my.fit.edu
Adam Hill ahill2013@my.fit.edu
Chris Kocsis ckocsis2007@my.fit.edu
William Nyffenegger wnyffenegger2013@my.fit.edu

Faculty Sponsor:
Dr. Marius Silaghi msilaghi@fit.edu

Contents

Introduction

As stated in other documents, the core software components on the IGVC robot will be as follows:

- Control
- Vision
- Pathfinding
- GUI
- Motor Control
- Simulation
- Messaging Framework

Each of these components must be tested first in isolation, then in communication with other "fake" modules, then in conjunction with other modules, then finally on the physical robot, running through real courses.

During **isolated testing**, we are not concerned with its ability to send or receive messages, we are simply evaluating its core functionality. That is, given accurate information, can it effectively perform its calculations, etc. as specified in the requirements document? For this phase, modules will act on hard-coded data, or data read from a file.

During **message testing**, we verify a module's ability to send and receive the information it needs with the use of testing modules which simulate the functionality of the modules upon which the tested module depends, and modules which depend on the functionality the tested module provides. During this phase, it is also important to create fake modules which crash, or send inconsistent data, to observe what the tested module does in conditions of partial system failure.

During **interop testing**, all modules are assumed to have been tested using the previous two methods, and are therefore trusted to fulfill their individual responsibilities. Emphasis will be placed primarily on ensuring two things. First, the modules must communicate using identical units and conventions. For example, consider the possibility that vision detects an obstacle four meters in front of the robot and one meter to the right, and sends that information to pathfinding, which interprets the same information as four and one feet. Second, modules must never form deadlock conditions of any kind.

During **field testing**, all modules are assumed to be fully functional with each other in "ideal" conditions. Field testing verifies that each module will must use only an appropriate amount of the robot's system resources. It may not be apparent until this stage whether each module's cost to the system is low enough that they can all fulfill their functions with low latency. This phase will also be necessarily characterized by a substantial increase in the amount of incorrect and inconsistent data, on account of vibrations, skid, and other real world conditions which might not be replicable in previous testing phases.

Note that the Interop and Field sections below deliberately have minimal content, as the procedure for testing each is largely the same. In Interop, we connect some or all of the modules together and observe whether behavior differs significantly from the Messaging phase by monitoring logged messages. In the Field Phase, we install the software on the robot and observe whether it continues to fulfill requirements, mostly through trial and error.

Control & IOP

Isolation N/A; module's functions all involve messaging

Messaging

Testing control and IOP capabilities within the vehicle is in effect testing the entire messaging system. Startup, logging, and state changes are all message based.

- All messages to all subsystems will be tested as a validation of the communication framework. A maximum latency will be established based on the reception of message acknowledgements on the server for standard messaging.
- Some messages will simply send file locations. These messages will be tested by opening the files and checking the fidelity of the contents of those files.
- Memory usage and latency from high volumes of traffic will be experimented with to establish upper bounds for logging frequency and command frequency

Interop

- The main tests on interoperability will focus on completing required handshakes and executing series of commands sent remotely.
- Test client will be written that connects and interacts on the specified ports in documents. This client will contain suites of commands for testing responses to series of commands as specified in the SAE JAUS documents.
- Commands include, but are not limited to:
    - Changing the speed and direction of the robot
    - Receiving information on position, speed, and direction of the robot
    - Information movement rates
    - Task the platform is executing (waypoint navigation, initialization, etc.)
- The test client will run on a machine separate from IOP & Control, and attempt to connect to the vehicle through:
    - Ethernet connection
    - Local Wi-Fi
    - Remote Wi-Fi

Field

Connect remotely in the field and execute tests on multiple networks. Field testing verify performance particularly with the logging of messages that are time sensitive

- Logging time sensitive information
- Starting and restarting the robot
- Recovering a subsystem upon failure

- Reinitializing runs
- Execute suite of interoperability tests including local waypoint navigation as specified in competition documents
- Allow takeover of robot by remote test client including commands on speed and direction

Vision

Isolation

- ZED specifications must be tested to verify manufacturer specifications
    - Recording images, Point Cloud(PCL), and depth data at varied distances
    - Parsing and interpreting actual recordings to find maximum recording range and utility
    - Field of vision manufacturer specifications must be verified
        - Recording image data at the peripherals where they should be recorded and should not and determine the field of vision
- Lighting conditions must be verified and tested for each algorithm to determine the range of utility at different times of the day
- Weather conditions, specifically rain can provide many problems and we must test and validate the maximum effectiveness of our camera while it is raining
- Line detection testing cases:
    - Single image of a straight, white paint line
    - Multiple sloping lines
    - Line next to an object
    - Line partially obscured by an object

Messaging

- Receives current location from Position
- Recalculates any obstacles or lines based on the updated position
- Transmits obstacle and line data to Pathfinding

Interop

- Testing of interoperability shall focus on the latency of the data sent by the interoperability subsystem

Field

- Testing the actual hardware and software detection of objects and lines and runtime
- Recording software output with corresponding pictures if possible to show where the lines and obstacles are to prove the system is or is not finding each set of potential obstacles

Pathfinding

Isolation

- Tests involve one-time pathfinding on static fully and perfectly known maps

- It is important to verify that pathfinding can consistently find a solution in a variety of situations. As development on the pathfinding algorithm progresses, the following scenarios will be tested, in order of increasing difficulty:
    - Open field, no obstacles
    - Sparsely distributed objects much larger than robot
    - Robot sized obstacles, with ~3 times the width of the robot between them
    - Lanes of obstacles that form pathways without gaps to pass through
    - Robot is facing walls on the front, left, and right, and cannot avoid collisions without backing up
- A GUI application will be developed, borrowing some of the GUI module's functionality, which will allow the user to design new scenarios and run pathfinding against them.

Messaging

- Messages will be sent to the Pathfinding module, and paths will be requested. Messaging stress tests will involve the following:
    - Many path requests in less than the time it takes to complete one full pathfinding calculation.
        - Consecutive requests should be ignored
    - Messages adding obstacles during pathfinding calculations
        - Obstacles will be queued for addition once the pathfinding computation completes

Interop

- Must communicate with Motor Control by sending commands that correctly navigate the robot to the same position in real life as it is in the map.
- Must receive updated obstacle information from Image Processing.
    - Should never have to wait for new data. Automatically rerun SBMPC as soon as possible.

Field

Visual test.

- Does the robot hit any obstacles or go over lines?
    - If so, then refinement of the code is needed.
- Will be mostly trial-and-error

GUI

Isolation

The majority of the GUI's testing takes place in the isolation phase.

- Scripts will be run on the GUI to load various kinds of maps. The GUI passes if:
    - The elements of the map are layered properly
        - For example, the robot itself should not be obscured by an obstacle in the event of a collision

    - Objects are scaled and rotated to reflect their relative positions on the map. This will be tested on canvases of variable size to verify compatibility with different screens
    - Maps are displayed quickly. There is no particular time threshold in mind, but a slow application will be very obviously visible. We expect rendering to happen nearly instantly, so this shouldn't be an issue

Messaging

- A publisher will send identical map information that was used in previous phase. The rendered maps should be identical.
- A subscriber will listen for messages sent as a result of button clicks. Latency on messages sent should be less than 0.1 second, which can be verified by storing a time object when the button is clicked, and compared to the current time on the receiving end.

Interop N/A

Field N/A

Messaging Framework

Isolation

- Basic subscriber and publisher tests in every language being used
    - Verify connection resilience
    - Verify ability to receive and decode messages
    - Verify ability to post messages to the server
    - Verify ability to acknowledge messages received
- Testing the messaging framework shall to include:
    - At minimum 1.5x the number of messages expected to test the throughput
    - Testing the encoding and decoding of every possible JSON message especially with null values to test JSON performance.
    - Testing logging and tracking acknowledgements sent on the server
    - Testing long term behavior to verify that the server does not save unacknowledged messages
    - Determining the precision of heartbeats set by the server
- Performance metrics to be recorded:
    - Amount of time a message takes to go from one sub-system to the server
    - Amount of time a messages takes to go from server to the destination sub-system
    - Memory usage of the server under load
    - CPU usage of the server under load

Messaging

Messaging testing shall focus on verifying each subsystem's client for subscribing and publishing messages.

This includes:

- The ability of every subsystem to send and receive every possible message it may send or receive shall be tested. Specifically the focus will be on latency between receiving a request for information and publishing that information.
- The ability of the subsystem to recover from bad messages shall be tested by inserting poor messages into message publishers.
- The ability of subsystems to send messages and receive messages from particular subsystems shall be tested using routing key and exchange combinations
- Integration testing with multiple pieces of hardware including devices from Florida State University
    - Connection from multiple devices
    - Heartbeats and timing with multiple devices
    - Interaction with routers

Interop N/A

Field N/A

Tools

Several software tools are being utilized to help test and debug the software suite. These help speed our development process by a significant amount. Tools and their description are as follows:

- **Apache Log4j 2 –** an asynchronous logging tool for Java. Log4j provides easy logging of your code, typically with just one simple line. It is asynchronous which means it runs in its own, separate thread, and does not impact performance of your program.
- **Doxygen –** a code documentation tool. Used to easily document code in nearly any language. Compiled into a html format and is easily navigable. Use for referencing pieces of previously written code.
- **Gradle –** build tool with cross-platform build capabilities. Gradle will build your code, run test cases, and automatically deploy it to your platforms using scripts, making the process streamlined.
- **IntelliJ IDEA –** a Java IDE. Provides many "intelligent" features that help with code development. Has support for many useful plugins, such as code coverage, version control and Gradle.