Simulation Driven Real-Time Testing Of a Railroad System

Milan E. Soklic, Ph.D.
Computer Science Department
Florida Gulf Coast University, Fort Myers, FL 33965
msoklic@fgcu.edu

Keywords: single-board computer, SBC, application programming interface, API, hazards, real-time operating system VxWorks, personal computer workstation, PCW

Abstract
There are situations where a system in mind needs to be tested before it is built. Using modeling and simulation users can apply realistic interactions with their synthetic environments representing real world.

In this study, the synthetic environment is a hardware railroad model representing a miniature of the actual size railroad. Simulation software, used to test the railroad system, consists of two software subsystems: a virtual railroad which simulates behavior of the hardware railroad model, i.e., the physical railroad, and the controller which is used to synchronize and control both the virtual and the physical railroads. As a result, the controller represents a prophet and analyst of the physical railroad behavior.

This article is logically divided into two parts. The first part gives an overview of equipment needed to test the railroad system; it describes major hardware and software components, their purpose, as well as their interconnections. The second part is focused on major design views of simulation software used to: simulate the physical railroad model, synchronize virtual and physical railroads, and supervise possible conditions that may lead to railroad hazards.

Introduction
Modeling and simulation is used by users to apply realistic interactions with their synthetic environments in lieu of real world. Simulation is increasingly used to improve efficiency and effectiveness in engineering development. Models and simulations can provide valuable quantitative information that can clear up uncertainty, focus attention on key elements of decision making, and help justify decisions. They help remove doubt, second guessing, and the unknowns associated with a development of a solution to a problem. If designed properly, models and simulations can identify important performance and cost parameters, which can be compared to solidify understanding of a problem. Real-time software modeling and simulation has to be applied throughout the whole process of software life cycle [7, 8].

1. EQUIPMENT FOR TESTING THE RAILROAD SYSTEM

Equipment for testing the railroad system is assembled around several commercially available hardware and software components. Figure 1 depicts three major subsystems and their logical connections representing a fully operational system. The major hardware components are: the railroad model, representing a miniature replica of the actual-size railroad; the single-board computer (SBC), hosting software for simulation and testing the railroad model; and personal computer workstation (PCW), hosting the Tornado integrated tool for software development running under real-time VxWorks operating system [1, 2]. The railroad model is pure hardware and contains no notion of a software program.

![Figure 1: The configuration of three subsystems into equipment for analysis and testing of a railroad system.](image)

Simulation software, i.e., the simulator, is first developed on PCW, in the Tornado operating environment, and then down loaded, via the Ethernet line, to the SBC. To gain real-time control of the hardware railroad model, the simulator running on SBC interacts with the hardware railroad model via the RS232 line, whereas the user communicates with the simulator via the keyboard and display of the PCW.

Figure 2 shows an example of a track layout of the hardware railroad model. There are eleven track-switches...
shown as octagons with numbers representing track-switch addresses, and 29 sensor-locations, shown as numbers along the tracks, used to detect train movements. The symbols N, S, E, and W represent the compass orientations of the track layout; they are used to describe the direction of a train movement. The trains are miniature replicas of trains used on actual railroads. All components for the railroad model are manufactured by Märklin Company [6].

Figure 2: The track layout of the hardware railroad model with 11 track-switches 29 sensor locations.

Trains, track-switches, and detection of train locations are controlled by the simulator via three hardware modules: the control module, the interface module, and the track-detection module. Conceptual organization of the three hardware modules is shown in Figure 3. The interface module is a gateway for sending commands from SBC to either the control module or to the track-detection module. Control signals, coming from the SBC to the hardware railroad model, are marked with bold arrow-lines. The control module broadcasts commands for locomotives and track-switches over the middle rail of the track, indicated with a dotted line. The track-detection module, independently of other modules, gathers data from all sensor locations along the track.

The railroad track has three rails; the middle rail is shown as a dotted line. Sensors can be placed either close to the two outer rails, or in the middle rail. A sensor location has one up to three sensors. The number of sensors in a sensor location depends on the test under study. Sensors can only detect moving trains in their close vicinity and cannot identify them. Thus, when a force emanating from the train triggers a sensor, the sensor built-in switch makes a momentary closure on its terminal, sets its corresponding track-detection module latch input to logic high, indicating a train triggered this sensor. Sensors’ data stored in the track-detection module are read by the simulator.

Figure 3: Principal arrangement of hardware modules in the software controlled railroad model.

2. SOFTWARE FOR SIMULATION AND TESTING OF THE RAILROAD

Software for the real-time railroad simulation and testing consists of two major modules: the virtual railroad module (VR), and the controller module (CM). The virtual railroad module simulates the physical railroad environment, i.e., it supports running of multiple virtual trains at the same time, such as train speed and train direction, as well as supports settings of virtual railroad track-switches. In short, the virtual railroad module replicates the functionality of the physical railroad. The controller module is used to control the railroad to avoid train collision and to perform the railroad testing. The virtual railroad and the controller together are referred to as the railroad simulator.

The controller module supports two running modes: virtual-only and virtual-and-physical mode. If virtual-only mode is used, the simulator runs only the virtual railroad, whereas if both modes are used simultaneously, the physical railroad is used to synchronize the virtual railroad, i.e., it mirrors physical railroad states on the virtual railroad.

Figure 4 shows the context diagram depicting the vital information flow in the high-level software architecture. The keyboard console is utilized for user to input data to the
Simulator in order to change properties of the virtual railroad, the virtual railroad then passes the user input data to the physical railroad thus sustaining synchronization with the physical railroad. For instance, the user can start, stop, and change speed and direction of all designated trains first on virtual railroad and then on the actual hardware railroad.

Figure 4: Context DFD (simulator)

On Figure 5 the level 1 data flow diagram describes the simulator in terms of high-level internal modules. The Controller Module (CM) interacts directly with both the virtual and physical railroads in the same manner. This design was chosen to remove cohesion between system components and support running the CM on either simulated or actual sensor data with little to no modification to the CM.

The output produced by the Virtual Railroad (VR) regarding the status of the virtual trains is sent to the PCW for display and user verifications purpose. The user can observe the output produced by the simulator and compare it with the status of the physical railroad.

When the simulator is operating in synchronization with the physical railroad, the simulator detects all hazardous conditions and executes the most optimal solution in parallel to both the virtual railroad and the physical railroad in order to avoid collision. For synchronization purposes, the simulator reads the physical track-detection module once every time a selected sensor location on the physical railroad has been triggered. Once this sensor has been triggered by a specific physical train, then the virtual train that corresponds to that train changes its position to that sensor location on the virtual railroad thus obtains synchronization. For instance, if train number 65 on the physical railroad triggers the sensor location 14 then the corresponding virtual train 65 on virtual track will also be changed to that position.

Figure 5: Level 1 DFD (simulator)

Figure 6 shows the level 2 data flow diagram representing an exploded view of the CM. The User Interface (UI) module is used to provide the current status of the physical and/or virtual railroads. The Collision Avoidance Logic Module (CALM) is used to collect virtual and/or physical railroad data and user commands and use these data items to predict whether two or more trains will collide. If the logic within this module determines that a collision will happen, it issues corrective commands to the virtual and/or physical railroads. The CALM is used to:

(i) accept train commands from the UI,
(ii) receive feedback data from the virtual/physical railroad via sensor polling (physical feedback) and timer alerts (virtual feedback),
(iii) process the current railroad and train states and detect/predict potential collisions,
(iv) implement collision avoidance logic, and issue train and railroad commands to avoid collisions.

The UI module accepts user text commands via the keyboard, and displays physical and/or virtual railroad status from the CALM. The Command Handler (CH) module accepts commands from the CALM and sends them to either the physical or virtual railroads, or both. The Sensor Monitor (SM) module monitors the physical railroad if running in physical or both modes, and provides sensor data to the CALM. The UI and SM run as separate tasks to avoid blocking the CALM.

Figure 7 shows the level 2 data flow diagram of the virtual railroad module. The Simulation Logic Module (SLM) simulates the physical railroad. The Timer module creates time delay, in seconds, and a Sensor object that 'fire' when the time delay expires. Each Timer instance gives two outputs: (i) to the CALM it outputs char array containing bits that correspond to the sensor that has been triggered, and (ii) to the SLM it outputs the number of the sensor location that has been triggered.
Figures 8.a and 8.b show modules and actual task names used in the implementation of the simulator. On Figure 8.a, it is seen that the SensorMonitor task periodically polls the physical railroad for updated sensor data. Once sensor data is found to be available, it notifies the CALM.

It can also be seen that the CALM will not be notified if no new data is available. That is the purpose of the self-checking CheckSensorData function defined within the SensorMonitor.

On Figure 8.b, it is seen that virtual railroad is not polled, it instead uses timers to simulate updated sensor data. The feedback of the timer to the Simulation-LogicModule is used to support the on-going loop of feedback-controlled logic within the virtual railroad.

2.1 Safety Hazards and Safety Measures
The internal checking via CALM takes charge of the following safety hazards:
(i) train collisions, e.g.: (a) two trains occupying the same segment at the same time, (b) a faster train overtaking a slower train, (c) two trains heading;
(ii) train entering turn at excessive rate of speed;
(iii) train running into an end-of-track bumper; and
(iv) sensor reading timeout.

All of these hazards, except the last, are high risk as they can cause physical damage to the train. These faults can arise from any of the following conditions, such as:
• user error in inputting train/track settings,
• CALM failure or overload, and
• loss of physical connectivity to the hardware railroad model.

The likelihood of a fault is fairly low if intelligent input handling is added to the UI module and CALM module that ensures that not only valid commands are entered but also
that only commands that do not affect the overall safety of the virtual railroad system are followed. If a fault does occur, the detection time is the time between physical railroad sensor reads, typically under one hundred milliseconds. Once the physical railroad sensors are read, the system can then react to the fault conditions that are present.

The safety measures put in place to alleviate the hazards are alarms and internal checking. Alarms are brought to the attention of the user in conditions, such as: sensor read time outs, message queue time outs, task spawn errors, and any exceptions that get raised.

The above listed hazards are very high risk and the virtual railroad system has no tolerance these hazards. The CALM constantly checks the current and future states to detect future collisions and hazard conditions. The CALM module, upon detecting a possible hazard, immediately issues commands to minimize exposure to this hazard, and to avoid other possible hazards. If a collision cannot be avoided by issuing commands, the railroad is shutdown immediately.

2.2 Implementation Specification

The simulation software was implemented in C++ programming language using object-oriented approach. The implementation structure of the virtual railroad is a combination of procedural and object-oriented code. For the most part, all internal data is represented as objects, and all most logical operations are handled procedurally. This allowed the greatest flexibility in encapsulating data while taking advantage of multi-tasking and shared memory capabilities of VxWorks environment. For this reason, control-flow functions were withdrawn from active objects and implemented via functions.

Figure 9: UML diagram (virtual railroad)
The use of exception handling is utilized in all software components, i.e. components in the simulation software throw exceptions when certain error conditions occur. These exceptions are handled by higher level components which either correct the condition or alert the user.

Figure 9 shows the object structures that have been created to model the hardware railroad and how they relate to each other. For instance, a Track object is composed of multiple Sections, which in turn, are composed of two Sensor objects. Using an object structure and object-oriented design for the railroad model allowed the developers to think in terms of the solution instead of getting mired in the details of the problem.

All error handling is implemented via C++ error handling support. Errors encountered by the Simulator software fall into two general categories: (1) errors that endanger the railroad, such as, unavoidable collision, (2) errors that don't put the railroad in immediate danger. Any errors of type (1) will result in the railroad being shutdown immediately as to avoid physically damaging the railroad. Errors of type (2) are seamlessly handled by CALM.

3. CONCLUSION
The simulation software is composed of a number of small procedural routines that comprise the task bodies, and a number of C++ objects that accurately model the physical railroad and its operation in software. The main tasks of the simulator are simulation of the railroad and collision avoidance systems, which run in parallel, simulating and predicting the future paths of all running trains.

The simulator uses software timers, implemented as C++ objects to model the event of a train passing over a physical sensor. These timers are created by the simulator and when they 'trigger', are consumed by both the simulator and the collision avoidance systems, allowing each to derive that railroad state it needs in order to function properly.

The collision avoidance system consumes the output of the timers and tries to forecast possible collisions, whether they be train-train collisions or train-bumper collisions. The collision avoidance system uses prediction logic to determine where the trains are and where they are going over a given period of time and uses these 'paths' to predict possible collisions. It then implements logic where it tries to figure out the best means to avoid the collision, whether it be setting a track-switch, accelerating a train, stopping a train, and sends the appropriate command to the simulated and physical railroads.

The collision avoidance algorithm uses a 'smart' look-ahead feature that only change the state of the trains on the railroad or the railroad itself if the change to be made won't create another collision. In other words, the change to alleviate the present collision condition itself will not create new collision conditions. This is accomplished by 'knowing' where the trains on the track will go in the next \( n \) seconds. Using these predicted paths, the collision avoidance algorithm can find possible collisions well before they are a danger to the trains themselves by comparing the paths of all trains on the track and flagging common way points in the paths. Once a possible collision has been detected, possible solutions are gathered and 'what-if' scenarios are generated for each of the solutions. These 'what-if' scenarios are simply predicted train paths that are themselves checked for possible collisions. If a generated 'what-if' scenario alleviates the current predicted collision and itself does not cause any collisions, then the action is taken, both on the physical and virtual railroads.

Being that this software system is indeed a real-time software system, real-time constructs such as semaphores, message queues, and signals have been used in its development. The semaphore construct is used throughout the system to control access to both the physical railroad and the virtual railroad, ensuring that race conditions cannot occur. Any components that need to communicate all utilize the message queue construct. The message queue gives flexibility in how software modules communicate and how these communications are handled by the recipient.

References